Z80

PROGRAMMATION DU

RODNAY ZAKS



RODNAY ZAKS

PROGRAMMATION

DU

Z80



Paris • Berkeley • Düsseldorf • Londres

POUR UN CATALOGUE COMPLET DE NOS PUBLICATIONS

FRANCE
6-8, Impasse du Curé
75881 PARIS CEDEX 18
Tél. : (1) 203.95.95
Télex : 211801

U.S.A.
2344 Sixth Street
Berkeley, CA 94710
Tel. : (415) 848.8233
Telex : 336311

ALLEMAGNE
Volgelsanger. WEG 111
4000 Düsseldorf 30
Post Bos N° 30.09.61
Tel. : (0211) 626441
Telex : 08588163

Traduction française : **Hubert Gayet, Mathieu Rossi**

Couverture de **Jean-François Pénichoux**

ISBN 2-7361-0058-4

(Edition originale : ISBN 0-89588-069-5, Sybex Inc., Berkeley)

TABLE DES MATIERES

LISTE DES ILLUSTRATIONS

PRÉFACE

Ce livre a été conçu comme un support complet et autonome de l'apprentissage de la programmation à l'aide du Z80. Il pourra être utilisé avec profit par un novice en matière de programmation, et en même temps, intéresser tous les utilisateurs du Z80.

A ceux qui disposent d'une expérience de la programmation, ce livre enseignera les techniques spécifiques utilisant les caractéristiques propres du Z80. Aux autres, il exposera l'ensemble des techniques indispensables pour commencer à programmer effectivement. Notre objectif est de fournir un niveau réel de compétence à qui souhaite programmer ce microprocesseur. Naturellement, aucun livre ne permet de faire l'économie de la pratique. Toutefois, nous souhaitons amener le lecteur au point où il se sentira capable de commencer à programmer lui-même, et à résoudre des problèmes simples, ou même modérément complexes, à l'aide d'un microordinateur.

Ce livre repose sur l'expérience de l'auteur qui a enseigné la programmation des microordinateurs à plus de mille personnes. Il s'agit, en conséquence, d'un ouvrage structuré, dont les chapitres présentent un degré de complexité croissant. Les lecteurs déjà familiarisés avec la programmation élémentaire pourront se dispenser du chapitre d'introduction. Par contre, les novices auront intérêt à lire plusieurs fois les derniers paragraphes de certains chapitres. Le livre a été conçu pour faire systématiquement avancer le lecteur à travers tous les concepts et techniques de base indispensables à la réalisation de programmes de complexité croissante. Il est donc recommandé de respecter l'ordre des chapitres. De plus, il est important que le lecteur s'efforce à résoudre le plus grand nombre possible d'exercices. Ces exercices, dont la difficulté a été soigneusement graduée, ont pour objectif de vérifier que les théories et techniques présentées ont été assimilées. Sans eux, il n'est pas possible de tirer le profit maximum de l'approche éducative de ce livre. Leur résolution pourra prendre du temps, dans certains cas, tel celui de la multiplication. Il s'agit là pourtant d'une expérience de programmation irremplaçable. Apprendre par la pratique, c'est indispensable.

Pour ceux qui auront pris goût à la programmation, d'autres livres de cette série abordent les différents aspects de la programmation, pour les microprocesseurs les plus répandus.

Pour une meilleure connaissance du matériel, on se reportera à deux ouvrages de référence : « Microprocesseurs » (C4) et « Techniques d'interface » (C5). Le contenu de cet ouvrage a été soigneusement vérifié et est donné pour fiable. Toutefois, des erreurs, typographiques notamment, pourraient éventuellement y être decelées. L'auteur accueillera avec reconnaissance toute remarque à ce sujet. Toute suggestion d'amélioration, concernant par exemple la possibilité de développement de programmes supplémentaires, sera également appréciée.

1

LES CONCEPTS DE BASE

INTRODUCTION

Ce chapitre introduit les concepts et définitions de base relatifs à la programmation des ordinateurs. Le lecteur auquel ces notions sont déjà familières, pourrait être tenté de se contenter d'un survol, avant d'aborder directement le chapitre 2. Toutefois, même le lecteur expérimenté est encouragé à lire ce chapitre introductif. Nombre de concepts importants y sont présentés : le complément à deux, le DCB, et d'autres modes de représentation. Certains peuvent encore constituer une nouveauté pour le lecteur. Les autres concepts devraient permettre aux programmeurs fussent-ils expérimentés, d'accroître leurs connaissances et leur habileté.

QU'EST-CE QUE LA PROGRAMMATION ?

Confronté à un problème, on doit d'abord s'efforcer d'élaborer une solution. L'expression de cette solution, sous forme d'une procédure à suivre étape par étape, s'appelle un *algorithme*. Un algorithme est la description pas à pas de la solution d'un problème posé. Il doit s'achever au terme d'un nombre fini d'étapes, et peut être exprimé dans n'importe quel langage ou représentation symbolique. Voici un exemple simple d'algorithme :

1. insérer la clé dans le trou de la serrure
2. tourner la clé d'un tour complet vers la gauche
3. saisir la poignée
4. tourner la poignée vers la gauche et pousser la porte.

A ce moment, si l'algorithme est correct pour le type de serrure concerné, la porte s'ouvrira. Cette procédure en quatre étapes constitue un algorithme d'ouverture de porte. Une fois la solution d'un problème exprimée sous forme d'un algorithme, celui-ci doit être exécuté par l'ordinateur. Malheureusement, il est maintenant bien connu que les ordinateurs ne comprennent, ni ne peuvent exécuter, les directives exprimées en français courant

(ou en toute autre langue humaine). Cela tient à *l'ambiguïté syntaxique* de tous les langages humains usuels. Seul un sous-ensemble bien défini d'un langage naturel peut être « compris » par l'ordinateur. C'est ce qu'on appelle un *langage de programmation.*

La conversion d'un algorithme en une suite d'instructions exprimées dans un langage de programmation s'appelle *programmation.* Pour être plus précis, la phase de traduction proprement dite de l'algorithme dans le langage de programmation constitue le *codage.* En fait, le terme de programmation désigne non seulement le codage, mais également la conception d'ensemble des programmes et des « structures de données » qui concrétiseront l'algorithme.

La programmation effective nécessite non seulement la compréhension des techniques possibles d'implémentation des algorithmes standards, mais aussi l'utilisation efficiente de toutes les ressources hardware — telles que registres internes, mémoires et organes périphériques. Elle suppose aussi une part de créativité dans l'utilisation de structures de données adaptées. Ces techniques seront décrites dans ce livre.

La programmation exige également une stricte discipline de documentation, de sorte que l'auteur des programmes ne soit pas le seul à pouvoir les comprendre. La documentation doit être à la fois interne et externe au programme.

La documentation interne d'un programme désigne les commentaires qui, placés dans le corps du programme, en expliquent le fonctionnement.

La documentation externe désigne les documents de conception extérieurs au programme : explications écrites, manuels et ordinogrammes.

LES ORDINOGRAMMES

Une étape intermédiaire est presque toujours respectée entre *l'algorithme* et le *programme* : c'est *l'ordinogramme.* Un ordinogramme n'est autre qu'une représentation symbolique de l'algorithme, sous forme d'une suite de rectangles et de losanges, contenant chacun une étape de l'algorithme. On utilise un rectangle pour les *ordres,* ou « instructions exécutables », et un losange pour les tests du type : Si l'information X est vraie, alors prendre l'action A, sinon l'action B. Plutôt que de donner ici une définition formelle des ordinogrammes, il nous paraît préférable de n'introduire et de ne discuter cette notion qu'ultérieurement, au fur et à mesure de la présentation des programmes.

Le tracé de l'ordinogramme est une étape intermédiaire qu'il est fortement conseillé de respecter, entre la spécification de l'algorithme et le codage proprement dit de la solution. Il est remarquable que, selon certaines observations, seuls quelques 10 % de l'ensemble des programmeurs sont capables d'écrire correctement un programme sans tracer l'ordinogramme. Malheureusement, d'autres observations révèlent que 90 % des programmeurs croient faire partie de ces 10 %. Conséquence : 80 % des programmes, en moyenne, ne fonctionnent pas lors de leur première exécution en machine (ces pourcentages n'ont bien sûr pas la

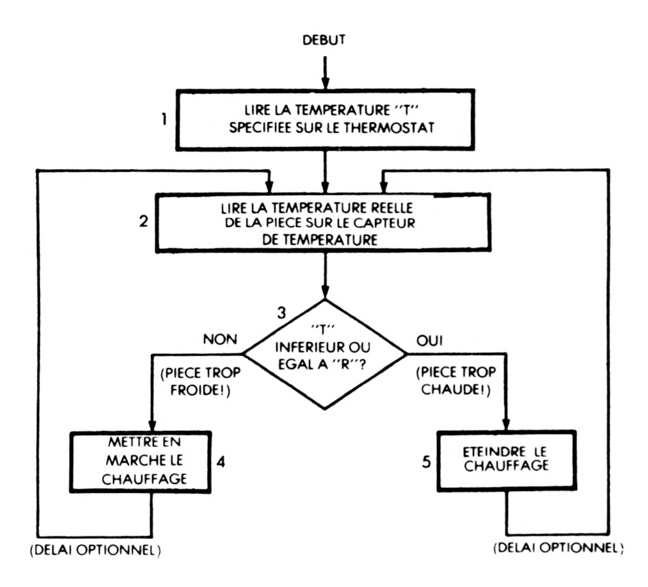


Figure 1.1. — Un ordinogramme pour maintenir constante la température d'une pièce

prétention d'être précis). En bref, la plupart des programmeurs novices voient rarement la nécessité de tracer un ordinogramme. Leurs programmes s'avèrent, très souvent, « boiteux » ou erronés, et ils doivent alors perdre beaucoup de temps à les tester et à les corriger (ce qu'on appelle la phase de *mise au point,* en anglais « debugging »). Il est donc vivement recommandé de s'astreindre, dans tous les cas, à la discipline qui consiste à tracer l'ordinogramme. Cela exige un petit travail supplémentaire avant le codage, mais a pour résultat habituel un programme clair, qui s'exécutera vite et correctement. Une fois bien comprise la technique de l'ordinogramme, seul un petit nombre de programmeurs est capable d'effectuer mentalement cette étape sans avoir recours au tracé sur le papier. Mais même dans ce cas, les programmes qui en résultent sont en général difficilement compréhensibles, sans la documentation que constitueraient les ordinogrammes, pour tout autre que leur auteur. En conséquence, il est universellement recommandé de s'imposer, pour tout programme important, la stricte discipline de tracer l'ordinogramme. De nombreux exemples illustreront la validité de ce principe, tout au long du livre.

LA REPRÉSENTATION DE L'INFORMATION

Tous les ordinateurs manipulent l'information sous forme de nombres ou sous forme de caractères. Nous examinerons maintenant les représentations internes et externes de l'information traitée par un ordinateur.

REPRÉSENTATION INTERNE DE L'INFORMATION

Toute information est gardée dans un ordinateur sous forme de groupes de bits. *Bit* est l'abréviation de l'anglais « binary digit », c'est à dire *chiffre binaire* (« 0 » ou « 1 »). En raison des limitations de l'électronique conventionnelle, la seule représentation pratique de l'information s'appuie sur une logique à deux états (la représentation de l'état « 0 » et de l'état « 1 »). Les deux états des circuits utilisés en électronique, digitale ou numérique, sont généralement « haut » et « bas ». Ils sont représentés par les symboles logiques « 1 » et « 0 ». Parce que ces circuits servent à réaliser des fonctions « logiques », on les appelle des circuits de « logique binaire ». La presque totalité du traitement de l'information est effectué actuellement en format binaire. Dans le cas général des microprocesseurs, et celui, particulier, du Z80, ces bits sont structurés en groupe de huit. Un groupe de huit bits s'appelle un *octet*. Un groupe de quatre bits s'appelle, lui, un *quartet*.

Examinons maintenant la manière dont l'information est représentée, de façon interne, dans ce format binaire. Deux entités doivent être représentées dans l'ordinateur. La première est le programme, qui est une suite d'instructions. La seconde est l'ensemble des données sur lesquelles le programme est appelée à travailler, et qui peut comprendre, à la fois des nombres et du texte alphanumérique. Nous examinerons les modes de représentations utilisés dans ces trois cas : programme, nombre et texte.

Représentation des programmes

Toutes les instructions sont représentées, de façon interne, par un ou plusieurs octets. Les instructions dites courtes le sont par un seul octet. Les instructions longues par deux octets ou plus. Le Z80, qui est un microprocesseur huit bits, va chercher les octets un à un en mémoire. Par suite, l'exécution d'une instruction composée d'un seul octet a toute chance d'être plus rapide que celle d'une instruction de deux ou trois octets. Nous verrons par la suite qu'il s'agit là d'une caractéristique importante du jeu d'instructions de tout microprocesseur, et en particulier du Z80, pour lequel un effort spécial a été entrepris, afin de fournir le plus grand nombre possible d'instructions composées d'un seul octet. Cela, bien sûr, pour améliorer l'efficacité d'exécution des programmes. Cependant, la limitation à une longueur de 8 bits implique d'importantes restrictions qui seront mises en évidence par la suite. Voilà, en tout cas, un exemple classique de compromis entre la rapidité et la souplesse de programmation. Le code binaire par lequel sont représentées les instructions est imposé par le constructeur. Le Z80, comme tout autre microprocesseur, n'échappe pas à la règle, et dispose d'un jeu fixe d'instructions. La liste en est donnée, avec le code correspondant, à la fin de ce livre. Tout programme sera exprimé en une suite de ces instructions binaires. Les instructions du Z80 sont présentées au chapitre 4.

Représentation des données numériques

La représentation des nombres n'est pas tout à fait évidente. Il est nécessaire de distinguer plusieurs cas : celui des nombres entiers, celui des nombres signés, autrement dit positifs et négatifs, et celui, enfin, des nombres décimaux. Passons en revue ces impératifs, et leurs solutions possibles.

Les entiers peuvent être représentés sous forme *binaire directe*. La représentation binaire directe n'est autre que l'expression, dans le système de numérotation binaire, de la valeur du nombre. Dans le système binaire, le bit le plus à droite représente 2 à la puissance zéro ; le bit placé immédiatement à sa gauche représente 2 à la puissance 1, le bit suivant 2 à la puissance 2, et ainsi de suite jusqu'au bit le plus à gauche, qui représente 2 à la puissance 7 = 128

$$b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0$$

représente

$$b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0$$

On appelle généralement b_0 le bit de plus faible poids, et b_7 le bit de plus fort poids, car ils sont respectivement associés à la plus petite et à la plus grande des puissances de 2 utilisées.

Les puissances de 2 sont :

$$2^7 = 128 \;;\; 2^6 = 64 \;;\; 2^5 = 32 \;;\; 2^4 = 16 \;;\; 2^3 = 8 \;;\; 2^2 = 4 \;;\; 2^1 = 2 \;;\; 2^0 = 1$$

La représentation binaire est analogue à la représentation décimale des nombres, dans laquelle « 123 » représente :

$$\begin{array}{rrrrr} & 1 \times & 100 & = & 100 \\ + & 2 \times & 10 & = & 20 \\ + & 3 \times & 1 & = & 3 \\ \hline & & & = & 123 \end{array}$$

Remarquons que $100 = 10^2$, $10 = 10^1$, $1 = 10^0$.

Dans cette notation positionnelle, chaque chiffre représente une puissance de 10, alors que dans le système binaire, chaque chiffre binaire, ou « bit », représente une puissance de 2.

Exemple : « 00001001 » représente :

$$\begin{array}{rrrl} 1 \times & 1 & = & 1\ (2^0) \\ 0 \times & 2 & = & 0\ (2^1) \\ 0 \times & 4 & = & 0\ (2^2) \\ 1 \times & 8 & = & 8\ (2^3) \\ 0 \times & 16 & = & 0\ (2^4) \\ 0 \times & 32 & = & 0\ (2^5) \\ 0 \times & 64 & = & 0\ (2^6) \\ 0 \times & 128 & = & 0\ (2^7) \\ \hline & & = & 9 \end{array}$$

en décimal

Examinons un autre exemple :

« 10000001 » représente

$$
\begin{array}{rcrcr}
1 & \times & 1 & = & 1 \\
0 & \times & 2 & = & 0 \\
0 & \times & 4 & = & 0 \\
0 & \times & 8 & = & 0 \\
0 & \times & 16 & = & 0 \\
0 & \times & 32 & = & 0 \\
0 & \times & 64 & = & 0 \\
1 & \times & 128 & = & 128 \\
\hline
 & & & = & 129
\end{array}
$$

en décimal = 129

« 10000001 » représente donc le nombre décimal 129.

En examinant la représentation binaire des nombres, il est aisé de comprendre pourquoi les bits sont numérotés de 0 à 7, de la droite vers la gauche. Le bit 0 est « b_0 » ; il correspond à 2^0 ; le bit 1 est « b_1 », et correspond à 2^1, et ainsi de suite.

DECIMAL	BINAIRE	DECIMAL	BINAIRE
0	00000000	32	00100000
1	00000001	33	00100001
2	00000010		
3	00000011		
4	00000100		
5	00000101	63	00111111
6	00000110	64	01000000
7	00000111	65	01000001
8	00001000		
9	00001001		
10	00001010	127	01111111
11	00001011	128	10000000
12	00001100	129	10000001
13	00001101		
14	00001110		
15	00001111		
16	00010000		
17	00010001		
		254	11111110
31	00011111	255	11111111

Figure 1.2. — Table décimal-binaire

La figure 1.2 montre les équivalents binaires des nombres de 0 à 255.

Exercice 1.1 : *Quelle est la valeur décimale de « 11111100 » ?*

Conversion décimal binaire.

Inversement, calculons l'équivalent binaire de « 11 » décimal.

11 ÷ 2 = 5 reste 1 → 1 (poids faible)
5 ÷ 2 = 2 reste *1* → *1*
2 ÷ 2 = 1 reste 0 → 0
1 ÷ 2 = 0 reste 1 → 1 (poids fort)

L'équivalent binaire de 11 est donc 1011 (nombre obtenu en lisant la colonne de droite de bas en haut).

L'équivalent binaire d'un nombre exprimé en décimal peut être obtenu à l'aide de divisions successives par 2, jusqu'à l'obtention d'un quotient nul.

Exercice 1.2 : *Quelle est l'expression binaire de 257 ?*

Exercice 1.3 : *Convertir 19 en binaire, puis reconvertir en décimal.*

Opérations sur les données binaires.

Les règles de l'arithmétique pour les nombres binaires sont évidentes. Les règles de l'addition sont les suivantes :

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = (1) 0

où (1) représente un « report » de 1 (remarquons que 10 est l'équivalent binaire de 2). La soustraction binaire s'obtiendra par addition du complément. Nous aurons l'occasion d'en expliquer le mécanisme lorsque nous aurons appris à représenter les nombres négatifs.

Exemple :

(2)	10
+ (1)	+ 01
= (3)	11

L'addition s'effectue exactement comme dans le système décimal, par addition sur les colonnes, de la droite vers la gauche.

Additionnons la colonne la plus à droite :

```
      10
    + 01
   ------
(0 + 1 = 1 pas de report)
```

Additionnons la colonne suivante :

```
      10
    + 01
   ------
      11        (1 + 0 = 1 pas de report)
```

Exercice 1.4 : *Calculer 5 + 10 en binaire ; vérifier que le résultat est 15.*

Quelques exemples supplémentaires d'addition binaire :

```
   0010 (2)        0011 (3)
 + 0001 (1)      + 0001 (1)
 ----------      ----------
 = 0011 (3)      = 0100 (4)
```

Ce dernier exemple illustre particulièrement le rôle du report.

En nous intéressant aux bits situés les plus à droite nous obtenons :

$$1 + 1 = (1)\,0$$

Un report de 1 est généré. Il doit être ajouté à la colonne de bits suivante.

```
   001 –  La colonne de droite vient juste d'être traitée
 + 000 –
 +   1  (report)
 -------
  (1) 0 –
```

Le résultat final est : 0100.

Un autre exemple :

```
   0111    (7)
 + 0011    (3)
 --------------
   1010 = (10)
```

Dans ce cas, un report est à nouveau généré. Il se propage jusqu'à la colonne située à l'extrême-gauche.

Exercice 1.5 : *Calculez le résultat de :*

```
   1111
 + 0001
 ------
 =    ?
```

Le résultat tient-il sur quatre bits ?

Avec huit bits, il est ainsi possible de représenter directement les nombres de « 00000000 » à « 11111111 », c'est-à-dire de 0 à 255. Deux obstacles apparaissent immédiatement. D'abord, seuls les nombres positifs sont représentés. Leur grandeur est, d'autre part, limitée à 255, si nous ne disposons que de huit bits. Traitons successivement chacun de ces problèmes.

Représentation binaire signée :

En représentation binaire signée, le bit situé le plus à gauche sert à indiquer le signe du nombre. Traditionnellement, « 0 » sert à signaler un nombre *positif,* et « 1 » un nombre *négatif.* C'est ainsi que « 11111111 » représente maintenant − 127, et « 01111111 » + 127. Il nous est dorénavant possible de représenter indifféremment des nombres positifs ou négatifs, mais la taille maximum de ces nombres a été ramenée à 127.

Exemple : « 0000 0001 » représente + 1 (le « 0 » de tête est le « + », suivi de « 000 0001 » = 1).

Exercice 1.6 : *Quelle est la représentation de « − 5 » en binaire signé.*

Intéressons-nous maintenant au problème de la *grandeur* des nombres. La représentation de nombres plus grands passe par l'utilisation d'un plus grand nombre de bits. Par exemple, en utilisant 16 bits (2 octets), nous pourrons traiter en représentation binaire signée des nombres allant de − 32 K à + 32 K (dans le jargon informatique 1 K représente 1024). Le bit 15 est utilisé pour le signe, et les 15 autres (du bit 14 au bit 0) pour la valeur absolue : 2^{15} = 32 K. Si cette grandeur s'avère encore insuffisante, on aura recours à 3 octets ou même davantage. On utilise, par exemple, un très grand nombre d'octets pour la représentation interne des très grands nombres entiers. C'est pourquoi la plupart des BASIC simples, et certains autres langages, n'offrent qu'une précision limitée. De cette façon, ils n'utilisent qu'un format interne court pour les entiers qu'ils manipulent. De meilleures versions de BASIC, ou d'autres langages, fournissent un plus grand nombre de chiffres significatifs, au prix de l'occupation, par chaque nombre, d'un plus grand nombre d'octets.

Autre problème : celui de l'efficacité et de la rapidité. Nous allons nous

efforcer d'effectuer une addition avec la représentation binaire signée que nous venons d'introduire. Ajoutons « − 5 » à « + 7 »

+ 7 se représente par	00000111
− 5 se représente par	10000101
La somme binaire est	10001100 soit − 12

Ce n'est pas le bon résultat, qui devrait être − 2. Pour pouvoir utiliser cette représentation, il est nécessaire d'entreprendre des traitements spéciaux, dépendant des signes, qui provoquent, toutefois, une complexité accrue et une réduction des performances. En d'autres termes, l'addition binaire des nombres signés ne fonctionne pas correctement. C'est très regrettable, car il est évident qu'un ordinateur ne doit pas se contenter de représenter l'information. Il doit aussi effectuer sur elle des opérations arithmétiques.

La solution de ce problème réside dans ce qu'on appelle la représentation en *complément à deux*, qui remplace avantageusement la représentation *binaire signée*. Son introduction nécessite cependant une étape intermédiaire : le *complément à un*.

Complément à un :

Dans cette méthode, tous les entiers positifs sont représentés en format binaire correct. Par exemple, « + 3 » est, comme d'habitude, représenté par 00000011. La représentation de son complément « − 3 » s'obtient en inversant chaque bit de la représentation initiale. Chaque 0 est transformé en 1, et chaque 1 en 0. Dans notre exemple, la représentation en complément à un de « − 3 » sera donc : 11111100.

Autre exemple :

+ 2 s'écrit 00000010
− 2 s'écrit 11111101

Remarquons que, dans cette représentation, les nombres positifs commencent, sur la gauche, par un « 0 », et les nombres négatifs par un « 1 ».

***Exercice 1.7 :** La représentation de « + 6 » est « 00000110 ». Quelle est la représentation de « − 6 » en complément à un ?*

Réalisons le test qui consiste à ajouter moins 4 à plus 6.

	− 4 s'écrit	11111011
	+ 6 s'écrit	00000110
la somme est	(1)	00000001,

où (1) représente un report.

Le résultat correct devrait être 2, soit 00000010.

Autre essai :

	− 3 s'écrit	11111100
	− 2 s'écrit	11111101
la somme obtenue est	(1)	11111001,

soit − 6, et un report. Le résultat correct est, en réalité, − 5 dont la représentation est 11111010. Autrement dit, ça ne marche pas.

Cette représentation permet de représenter des nombres positifs et négatifs. Cependant, leur addition ne fournit pas toujours un résultat correct. Nous sommes donc contraints de faire appel à une autre représentation, déduite du complément à un : le complément à deux.

Représentation en complément à deux :
Dans la représentation en complément à deux, de la même manière qu'en complément à un, les nombres positifs se représentent en binaire signé habituel. La différence réside dans la représentation des *nombres négatifs.* La représentation en complément à deux d'un nombre négatif s'obtient en calculant d'abord le complément à un, puis en *ajoutant un.* Exemple : + 3 se représente en binaire signé par 00000011. Son complément à un s'écrit 11111100. Le complément à deux s'obtient en ajoutant 1. Soit donc 1111101.

Essayons une addition :

(3)	00000011
+ (5)	00000101
= (8)	00001000

Le résultat est correct.
Essayons une soustraction :

(3)	00000011
(− 5)	11111011
	= 11111110

Identifions le résultat, en calculant le complément à deux.

Le complément à un de 11111110 est	00000001
on ajoute 1	+ 1
Le complément à deux est donc	00000010, soit + 2.

Notre résultat précédent, « 11111110 », représente − 2. Le résultat est correct.
Ainsi donc, les résultats obtenus lors d'une addition et d'une soustraction sont exacts (en ignorant le report). Il semble que la méthode soit bonne.

Exercice 1.8 : *Quelle est la représentation en complément à deux de « + 127 » ?*

Exercice 1.9 : *Quelle est la représentation en complément à deux de « − 128 » ?*

Additionnons maintenant + 4 et − 3 (la soustraction s'effectue en ajoutant le complément à deux) :

	+ 4 s'écrit	00000100
	− 3 s'écrit	11111101
Le résultat est :	(1)	00000001

En ne tenant pas compte du report, on obtient 00000001, soit « 1 » en décimal. Il s'agit là du bon résultat. Sans pouvoir ici en donner la démonstration mathématique complète, nous affirmons que cette représentation convient. Avec la méthode du complément à deux, il est possible d'additionner ou de soustraire des nombres signés, sans se préoccuper de leur signe. En utilisant les règles courantes de l'addition binaire, on obtient le résultat correct, signe compris. Le report est simplement ignoré. C'est un avantage très important. Si tel n'était pas le cas, il faudrait à chaque fois effectuer un traitement spécial, dépendant des signes des opérandes, ce qui augmenterait sensiblement la durée des opérations.

Pour être complet, disons que le complément à deux est simplement la représentation la plus commode, dans le cas de processeurs simples tels que les microprocesseurs. Dans les processeurs complexes, d'autres représentations sont parfois utilisées. Par exemple, le complément à un est utilisable, mais il nécessite un mécanisme spécial de correction du résultat.

Dorénavant, tous les entiers signés seront implicitement représentés, de façon interne, en notation en complément à deux. La figure 1.3. donne une table des nombres en complément à deux.

Exercice 1.10 : *Quels sont le plus petit et le plus grand nombres représentables en notation en complément à deux sur un seul octet ?*

Exercice 1.11 : *Calculez le complément à deux de « 20 », puis le complément à deux de votre résultat. Trouvez-vous de nouveau 20 ?*

Les exemples suivants vont nous permettre de mettre en évidence les règles associées au complément à deux. En particulier, C témoignera d'une éventuelle condition de report ou de retenue (c'est le bit 8 du résultat). V témoignera d'un débordement, c'est-à-dire d'un changement accidentel de signe, provoqué par la manipulation de nombres trop grands. Il s'agit, schématiquement, d'un report interne du bit 6 vers le bit 7 (le bit de signe). Le principe en sera explicité par la suite.

Mettons maintenant en évidence le rôle du report C (de l'anglais « Carry ») et du débordement V (de l'anglais « Overflow »).

+	code en complément à 2	−	code en complément à 2
+ 127	01111111	− 128	10000000
+ 126	01111110	− 127	10000001
+ 125	01111101	− 126	10000010
. . .		− 125	10000011
		. . .	
+ 65	01000001	− 65	10111111
+ 64	01000000	− 64	11000000
+ 63	00111111	− 63	11000001
. . .		. . .	
+ 33	00100001	− 33	11011111
+ 32	00100000	− 32	11100000
+ 31	00011111	− 31	11100001
. . .		. . .	
+ 17	00010001	− 17	11101111
+ 16	00010000	− 16	11110000
+ 15	00001111	− 15	11110001
+ 14	00001110	− 14	11110010
+ 13	00001101	− 13	11110011
+ 12	00001100	− 12	11110100
+ 11	00001011	− 11	11110101
+ 10	00001010	− 10	11110110
+ 9	00001001	− 9	11110111
+ 8	00001000	− 8	11111000
+ 7	00000111	− 7	11111001
+ 6	00000110	− 6	11111010
+ 5	00000101	− 5	11111011
+ 4	00000100	− 4	11111100
+ 3	00000011	− 3	11111101
+ 2	00000010	− 2	11111110
+ 1	00000001	− 1	11111111
+ 0	00000000	− 0	00000000

Figure 1.3. — Table des compléments à 2

Le report C

Voici un exemple de report (sur des entiers en binaire direct) :

$$
\begin{array}{rr}
(128) & 10000000 \\
+\ (129) & 10000001 \\
\hline
(257) = (1) & 00000001
\end{array}
$$

où (1) indique un report.

Le résultat rend nécessaire la présence d'un neuvième bit (le bit « 8 », puisque le bit le plus à droite est « 0 »). C'est le bit de report.

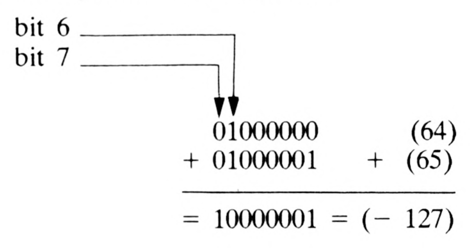
Si nous admettons que le report est le neuvième bit du résultat, ce dernier s'établira comme suit : 100000001 = 257.

Cependant, le report doit être détecté et traité avec soin. Dans le microprocesseur, les registres utilisés pour contenir l'information ont généralement une taille de huit bits. Lors du rangement du résultat, seuls les bits de 0 à 7 seront préservés.

Dès lors, un report nécessite toujours une action particulière : il doit être détecté par des instructions spéciales, avant d'être traité. Traiter le report équivaut soit à le ranger quelque part (à l'aide d'une instruction spéciale), soit à l'ignorer, soit enfin à décréter qu'il y a une erreur (si le plus grand résultat autorisé est « 11111111 ».).

Le débordement V

Voici un exemple de débordement :



Un report interne a été généré du bit 6 vers le bit 7. C'est ce que l'on appelle un débordement.

Le résultat est maintenant négatif, « par accident ». Cette situation doit être détectée, afin d'y remédier.
Examinons une autre situation :

```
      11111111     (− 1)
    + 11111111 + (− 1)
  ---------------------
  = (1) 11111110 = (−2)
      ↓
    report
```

Dans ce cas, un report interne a été généré du bit 6 vers le bit 7, et également du bit 7 vers le bit 8 (le report C défini au paragraphe précédent). Les règles de l'arithmétique en complément à deux établissent que ce report doit être ignoré. Le résultat est alors correct, puisque le report du bit 6 vers le bit 7 ne change pas le bit de signe.

Il ne s'agit pourtant pas là d'une condition de *débordement*. Lorsqu'on opère sur des nombres négatifs, le débordement ne se limite pas à un simple report du bit 6 vers le bit 7.

Examinons un autre exemple :

```
          11000000 (-  64)
          10111111 (-  65)
    ------------------------
    = (1) 01111111 (+ 127)
       ↓
     report
```

Cette fois, il n'y a pas eu de report interne du bit 6 vers le bit 7, mais report vers l'extérieur. Le résultat est incorrect, puisque le bit 7 (le bit de signe) a changé. Une condition de débordement doit donc être signalée.

Le débordement peut se produire dans quatre situations :

1. En additionnant de grands entiers positifs.
2. En additionnant de grands entiers négatifs.
3. En soustrayant un grand entier positif d'un grand entier négatif.
4. En soustrayant un grand entier négatif d'un grand entier positif.

Améliorons notre définition du débordement.

Techniquement, le témoin de débordement est un bit spécial réservé à cet usage, appelé « indicateur ». Il sera positionné dans le cas d'un report du bit 6 vers le bit 7 sans report vers l'extérieur, ou inversement, dans le cas d'un report vers l'extérieur non issu d'un report du bit 6 vers le bit 7. Voilà l'indication que le bit 7, c'est-à-dire le signe du résultat, a été changé accidentellement. Pour le lecteur féru de technique, signalons que l'indicateur de débordement est positionné par le OU exclusif des reports vers, et depuis, le bit 7 (le bit de signe). En pratique, chaque microprocesseur dispose d'un indicateur spécial pour détecter automatiquement cette condition, qui nécessite une action corrective.

Le report et le débordement.

Les bits de report et de débordement sont appelés « indicateurs ». Chaque microprocesseur en dispose, et nous apprendrons dans un prochain chapitre à les utiliser de façon à programmer efficacement. Ces deux indicateurs sont situés dans un registre spécial appelé registre d'indicateur, ou « registre d'état ». Ce registre contient également d'autres indicateurs, dont les fonctions seront explicitées dans le chapitre 4.

Exemples

Nous allons illustrer le fonctionnement du report et du débordement par des exemples concrets. Dans chaque cas, le symbole V désignera le débordement et C le report.

Si le débordement n'a pas lieu, V vaut 0. S'il a lieu, V vaut 1.

De même, pour le report C. Souvenons-nous que les règles de calcul en complément à deux spécifient de ne pas tenir compte du report (ce dont nous ne donnerons pas ici la démonstration mathématique).

Positif-Positif

```
  00000110 (+ 6)
+ 00001000 (+ 8)
-----------------
= 00001110 (+ 14)      V : 0     C : 0
```

(CORRECT)

Positif-Positif avec débordement

```
  01111111 (+ 127)
+ 00000001 (+   1)
------------------
= 10000000 (- 128)     V : 1     C : 0
```

Le résultat ci-dessus est incorrect puisqu'un débordement s'est produit (ERREUR)

Positif-Négatif (résultat positif)

```
       00000100 (+ 2)
+      11111110 (- 2)
---------------------
=(1) 00000010 (+ 2)    V : 0     C : 1 (ignoré)
```

(CORRECT)

Positif-Négatif (résultat négatif)

```
  00000010 (+ 2)
+ 11111100 (- 4)
----------------
= 11111110 (- 2)       V : 0     C : 0
```

(CORRECT)

Négatif-Négatif

```
       11111110 (- 2)
+      11111100 (- 4)
---------------------
=(1) 11111010 (- 6)    V : 0     C : 1 (ignoré)
```

Négatif-Négatif avec débordement

```
      10000001 (- 127)
+     11000010 (-  62)
  ----------------------
= (1) 01000011 (   67)    V : 1    C : 1
```

(ERREUR)

Cette fois, un débordement s'est produit, lors de l'addition de deux grands nombres négatifs. Le résultat devrait être − 189, nombre trop grand pour tenir sur huit bits.

Exercice 1.12 : *Complétez les additions suivantes. Indiquez le résultat, le report C, le débordement V, et précisez si le résultat est correct ou non.*

```
  10111111    (——)            11111010    (——)
+ 11000001    (——)          + 11111001    (——)
 ----------                  ----------
= ————————    V :——  C :——  = ————————    V :——  C :——
□ CORRECT     □ ERREUR      □ CORRECT     □ ERREUR

  00010000    (——)            01111110    (——)
+ 01000000    (——)          + 00101010    (——)
 ----------                  ----------
= ————————    V :——  C :——  = ————————    V :——  C :——
□ CORRECT     □ ERREUR      □ CORRECT     □ ERREUR
```

Exercice 1.13 : *Pouvez-vous donner un exemple de débordement en cas d'addition d'un nombre positif et d'un nombre négatif ? Pourquoi ?*

Représentation en format fixe

Nous connaissons maintenant le mode de représentation des entiers signés. Mais le problème de la taille des nombres reste en suspens.

Plusieurs octets doivent être utilisés pour représenter des entiers de grande dimension. Pour effectuer efficacement les opérations arithmétiques, il est nécessaire d'avoir recours à un nombre fixe d'octets, de préférence à un nombre variable. Une fois ce nombre choisi, la valeur du plus grand nombre pouvant être représenté est déterminée.

Exercice 1.14 : *Quels sont le plus grand et le plus petit nombre pouvant être représentés sur deux octets en complément à deux ?*

Le problème de la taille.

Lors des opérations d'addition, nous nous sommes limités à des nombres de huit bits, parce que le processeur utilisé opère, de façon interne, sur huit

bits à la fois. Autrement dit, nous n'avons pris en considération que les nombres contenus dans l'intervalle qui va de − 128 à + 127. Il est évident que cette restriction rend impossible de nombreuses applications. La précision multiple sera utilisée pour accroître le nombre de chiffres de la représentation. On peut utiliser des formats de deux, trois ou N octets, correspondant respectivement à la double, triple, Nuple précision.

Examinons, par exemple, un format double précision (16 bits)

00000000	00000000	est « 0 »
00000000	00000001	est « 1 »
. . .		
01111111	11111111	est « 32767 »
11111111	11111111	est « − 1»
11111111	11111110	est « − 2 »

Exercice 1.15 : *Quel est le plus grand entier négatif que l'on puisse représenter en complément à deux, avec un format triple-précision ?*

Cependant, cette méthode présente certains inconvénients. Lors de l'addition de deux nombres, il est généralement nécessaire d'opérer par groupes de huit bits. Nous y reviendrons au chapitre 3 (Techniques de base de programmation). Une telle méthode a pour conséquence un traitement plus lent. De plus, elle contraint à utiliser systématiquement 16 bits, même dans le cas où huit bits suffiraient à représenter le nombre. Pour cette raison, il est courant d'utiliser 16, voire 32 bits, mais rarement plus.

Considérons le point important suivant : quel que soit le nombre N de bits choisi pour la représentation en complément à deux, ce nombre est figé. Si un résultat, un calcul intermédiaire, quel qu'il soit, produit un nombre faisant appel à plus de N bits, certains bits vont être perdus. Le programme ne conserve normalement que les N bits les plus à gauche (les plus significatifs), et abandonne ceux dont le poids est faible. C'est ce qu'on appelle tronquer le résultat.

Voici, dans le système décimal, un exemple utilisant une représentation à six chiffres.

```
  123456
×    1.2
--------
  246912
 123456
--------
= 148147.2
```

Le résultat nécessite sept chiffres ! Le "2" situé derrière la virgule n'est pas conservé, et le résultat final sera 148 147. Il a donc été tronqué. Ordinairement, tant que la position de la virgule n'est pas perdue, cette

méthode est utilisée pour accroître l'étendue des opérations qu'il est possible d'effectuer au détriment de la précision.

Le problème est le même en binaire. La multiplication binaire sera exposée en détails au chapitre 4.

Cette représentation en format fixe peut être à l'origine d'une perte de précision, mais elle est en général suffisante pour les calculs mathématiques usuels. Malheureusement, dans le domaine de la comptabilité, nulle perte de précision ne peut être tolérée. On imagine mal, par exemple, dans un magasin où un client vient de faire de nombreux achats, que la somme à payer totalisée par la caisse enregistreuse soit limitée à trois chiffres et arrondie au franc voisin. Une autre représentation doit être utilisée chaque fois que la précision du résultat est essentielle. La solution couramment utilisée est le *DCB,* ou décimal codé binaire.

Représentation DCB

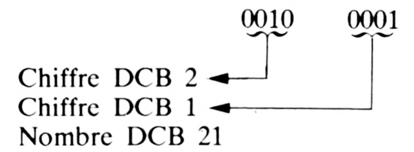
Le principe adopté pour représenter des nombres en DCB est de coder chaque chiffre décimal séparément, et d'utiliser autant de bits qu'il est nécessaire à une représentation complète du nombre. Pour coder chaque chiffre de 0 à 9, quatre bits sont nécessaires. Trois bits n'offriraient qu'une gamme de huit combinaisons, et se révéleraient donc incapables de représenter les dix chiffres. Quatre bits, autorisant seize combinaisons, sont par contre suffisants. Remarquons que six des codes possibles ne seront pas utilisés dans la représentation DCB (cf. fig. 1.4). Il y a là un problème potentiel, qui apparaîtra ultérieurement, lorsque nous serons amenés à résoudre certaines opérations d'addition et de soustraction. Dans la mesure où quatre bits seulement sont nécessaires pour coder un chiffre DCB, il est possible de ranger deux chiffres DCB dans un octet. C'est le *DCB compacté.*

CODE	SYMBOLE DCB	CODE	SYMBOLE DCB
0000	0	1000	8
0001	1	1001	9
0010	2	1010	inutilisé
0011	3	1011	inutilisé
0100	4	1100	inutilisé
0101	5	1101	inutilisé
0110	6	1110	inutilisé
0111	7	1111	inutilisé

Figure 1.4. — Table DCB

Ainsi « 0000 0000 » représente « 00 » en DCB et « 10011001 » représente « 99 ».

Un code DCB doit être lu comme suit :

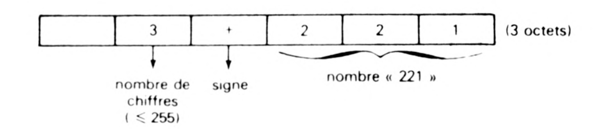


***Exercice 1.16 :** Quelle est la représentation DCB de 29, de 91 ?*

***Exercice 1.17 :** 1010 0000 est-elle une représentation DCB valide ? Pourquoi ?*

On utilise autant d'octets que nécessaires pour représenter tous les chiffres DCB. Généralement, un ou plusieurs quartets en tête de la représentation, servent à spécifier le nombre total de quartets, c'est-à-dire le nombre total de chiffres DCB. Un autre quartet, ou un octet, sert à indiquer la position de la virgule. Toutefois, les conventions peuvent varier.

Voici un exemple de représentation d'entiers sur plusieurs octets :



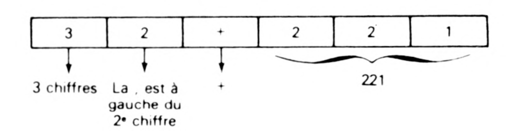
Cet exemple représente le nombre 221 (le signe + peut être représenté, par exemple, par 0000, et le signe − par 0001).

***Exercice 1.18 :** En utilisant la même convention, représentez − 23123 dans le format DCB ci-dessus, puis en binaire.*

***Exercice 1.19 :** Représentez en DCB 222 et 111, puis le résultat de 222 × 111 (calculez le résultat à la main, puis présentez-le dans le format ci-dessus).*

La représentation DCB s'adapte facilement aux nombres décimaux.

Par exemple, + 2,21 peut se représenter par :



L'avantage du DCB est de fournir des résultats absolument corrects. Son inconvénient est d'utiliser une grande quantité de mémoire, et d'entraîner des opérations arithmétiques lentes. Ceci est, à la rigueur, acceptable dans un environnement de comptabilité, mais généralement pas dans les autres.

Exercice 1.20 : *Combien de bits sont nécessaires pour coder 9999 en DCB ? et en complément à deux ?*

Nous avons maintenant résolu les problèmes liés à la représentation des entiers, et à celle des entiers signés, fussent-ils de grande taille. Nous avons exposé une méthode capable de représenter les nombres décimaux, avec la représentation en DCB. Examinons maintenant le problème de la représentation des nombres décimaux en format de longueur fixe.

Représentation en virgule flottante

Le principe de base est de représenter les nombres décimaux suivant un format fixe. La représentation *normalise* tous les nombres, de façon à éviter un gaspillage des bits. Par exemple, écrire « 0,000123 » revient à gaspiller trois zéros à la gauche du nombre. Ces chiffres n'ont, en effet, d'autre signification que de préciser la position de la virgule. La normalisation de ce nombre donne : $0{,}123 \times 10^{-3}$. « ,123 » est appelée *mantisse normalisée,* et − 3 *exposant.* Pour normaliser ce nombre, nous avons éliminé tous les zéros superflus situés à sa gauche, et ajusté l'exposant en conséquence.

Considérons un autre exemple :

22,1 se normalise en $,221 \times 10^2$.

Sous la forme générale :

$M \times 10^E$ où M est la mantisse et E l'exposant.

On aperçoit aisément qu'un nombre normalisé se caractérise, s'il est différent de zéro; par une mantisse inférieure à 1 et supérieure ou égale à 0,1.
En termes mathématiques, on écrirait :

$$0{,}1 \leq M < 1 \text{ ou } 10^{-1} \leq M < 10^{0}$$

De même, en représentation binaire :

$$2^{-1} \leq M < 2^{0} \text{ (ou } 0{,}5 \leq M < 1)$$

M étant la valeur absolue de la mantisse (sans tenir compte du signe).
Par exemple :

$$111{,}01 \text{ se normalise en : } ,11101 \times 2^3$$

La mantisse est 11101, et l'exposant 3.

Ayant défini le principe de la représentation, examinons le format réel. La figure 1.5 montre une représentation virgule flottante type.

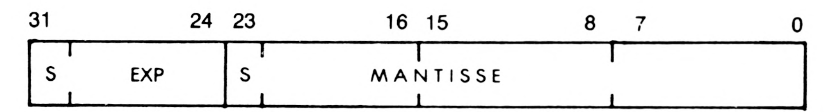


Figure 1.5. — Représentation virgule flottante type

Dans cet exemple, quatre octets sont utilisés, soit un total de 32 bits. Le premier octet, sur la gauche de la figure, sert à représenter l'exposant. Ce dernier, de même que la mantisse, sont représentés en complément à deux. Ainsi, l'exposant est-il compris entre − 128 et + 127. S, dans la figure 1.5. indique un bit de signe.

Trois octets sont utilisés pour représenter la mantisse. Comme le premier bit de la représentation en complément à deux indique le signe, il reste 23 bits pour représenter la valeur absolue de la mantisse.

Exercice 1.21 : *Combien de chiffres décimaux peuvent-ils être représentés avec une mantisse de 23 bits ?*

Ce n'est ici qu'un exemple de représentation en virgule flottante. Il est possible de n'utiliser que trois octets, ou au contraire, d'en utiliser davantage. La représentation sur quatre octets proposée ci-dessus n'est qu'une solution possible, parmi les plus courantes. Elle offre un compromis raisonnable entre la précision, la taille maximum des nombres, l'occupation mémoire, et l'efficacité du calcul.

Après les problèmes liés à la représentation des nombres, notamment des nombres entiers, signés ou non, et des nombres décimaux, occupons-nous de la représentation interne de données alphanumériques.

Représentation de données alphanumériques

La représentation de données alphanumériques, autrement dit de caractères, est tout à fait évidente : tous les caractères sont codés sur huit bits. Deux codes seulement sont d'usage courant dans le monde des ordinateurs : le code ASCII et le code EBCDIC. ASCII est le sigle d'une expression signifiant « Code Standard Américain pour l'échange d'information ». Il est universellement utilisé dans le domaine des microprocesseurs. L'EBCDIC est une variante du code ASCII utilisée par IBM. Il n'est donc pas utilisé dans le domaine des microprocesseurs, sauf lors de l'interfaçage d'un terminal IBM.

Examinons rapidement le système ASCII. Le codage des 26 lettres de l'alphabet, tant en majuscules qu'en minuscules, celui de 10 symboles numériques, et même de 20 symboles spéciaux supplémentaires, peut être aisément réalisé à l'aide de 7 bits, autorisant 128 codes possibles (voir fig. 1.6). Le huitième bit, lorsqu'il est utilisé, est le *bit de parité.* La technique de contrôle dite « de parité » permet de vérifier que le contenu d'un octet n'a pas été accidentellement changé. On compte le nombre de 1 dans l'octet. Le huitième bit est alors mis à 1, si ce total est impair. Cela s'appelle la parité paire. On peut aussi utiliser la parité impaire, autrement dit donner au huitième bit (celui le plus à gauche) une valeur telle que le nombre de 1 dans l'octet soit impair.

Exemple : calculons le bit de parité de 0010011 en parité paire. On dénombre trois 1 dans l'octet. Le bit de parité doit être 1 pour que ce total soit égal à 4 (donc pair). Le résultat est donc 10010011, le premier 1 représentant le bit de parité, et 0010011 identifiant le caractère.

La figure 1.6 montre la table des codes ASCII sur 7 bits. En pratique, on l'utilise soit telle quelle, c'est-à-dire sans parité, en ajoutant un 0 en huitième position (la plus à gauche), soit avec parité, en ajoutant à gauche le bit complémentaire approprié.

HEX poids faibles	poids forts / BITS	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	0000	NUL	DLE	SPACE	0	@	P	–	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB		7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	_ _
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	.	>	N	Λ	n	~
F	1111	SI	US	/	?	O	←	o	DEL

Figure 1.6. — Table de conversion ASCII
(voir l'appendice B pour les abréviations).

Exercice 1.22 : *Calculez la représentation sur huit bits des chiffres de 0 à 9, en parité paire (ce code sera utilisé dans les exemples pratiques du chapitre 8).*

Exercice 1.23 : *Même chose pour les lettres « A » à « F ».*

Exercice 1.24 : *En utilisant un code ASCII sans parité, (le bit le plus à gauche étant 0), indiquez le contenu binaire des quatre octets suivants.*

"A" "=" "-" "1"

Dans certains cas particuliers, télécommunications par exemple, d'autres codes peuvent être utilisés, tels que les codes autocorrecteurs. Ils sont, cependant hors du propos de ce livre.

Nous avons passé en revue les méthodes habituelles de représentation du programme et des données dans un ordinateur. Examinons maintenant les représentations externes possibles.

REPRÉSENTATION EXTERNE DE L'INFORMATION

Le terme de représentation externe désigne la manière dont l'information est présentée à *l'utilisateur,* c'est à dire généralement au programmeur. L'information est essentiellement présentée à l'extérieur sous trois formats : binaire, octal ou hexadécimal et symbolique.

1. *Binaire*

Nous avons vu que l'information est rangée de manière interne en octets, constitués chacun par une suite de huit bits (0 ou 1). Il est parfois souhaitable d'exposer cette information interne directement sous sa forme binaire ; c'est ce qu'on appelle la *représentation binaire.* Les diodes Electro-luminescentes (LED) (sortes de lampes miniatures) du panneau de commandes d'un microordinateur sont un exemple simple d'une telle représentation. Dans le cas d'un microordinateur huit bits, le panneau de commande sera généralement équipé de huit LED destinées à l'affichage du contenu d'un registre interne. Les registres internes, que nous décrirons au chapitre 2, servent à ranger des informations sur huit bits. Une LED allumée indique un 1, une LED éteinte un zéro. Une telle représentation binaire peut servir à la mise au point fine d'un programme complexe, surtout s'il comporte des entrées-sorties ; mais elle est bien sûr peu pratique au niveau humain. Dans la plupart des cas, en effet, il est préférable de percevoir l'information sous forme symbolique. Ainsi, est-il plus facile de comprendre ou de se souvenir d'un « 9 » que de « 1001 ». Des représentations plus appropriées ont été développées, pour améliorer la communication homme-machine.

2. *Octal et Hexadécimal*

En octal et en hexadécimal, un symbole unique représente, respectivement, trois et quatre bits. Dans le système octal, toute combinaison de trois bits est représentée par un chiffre compris entre 0 et 7.

Binaire	**Octal**
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Figure 1.7. — Symboles octaux

Par exemple, le binaire « 00 100 100 » se représente par
▼ ▼ ▼
0 4 4

Soit « 044 » octal.

Autre exemple : « 11 111 111 » s'écrit

3 7 7

Soit « 377 » octal.

Inversement, le nombre octal « 211 » représente

010 001 001

Soit « 10001001 » binaire.

L'octal était, traditionnellement, employé sur les anciens ordinateurs, qui utilisaient diverses tailles de mot, et un nombre de bits variant de 8 à 64, selon la machine. Plus récemment, avec le déferlement des microprocesseurs, la standardisation s'est faite sur un format de huit bits. Une autre représentation, plus pratique, s'est imposée : *l'hexadécimal.*

Dans la représentation hexadécimale, un groupe de quatre bits est codé sur un seul symbole hexadécimal. Les chiffres hexadécimaux sont représentés par les symboles 0 à 9, et par les lettres A, B, C, D, E, F. Par exemple, « 0000 » est représenté par 0, « 0001 » par 1, et 1111 par la lettre F (cf. fig. 1.8).

DECIMAL	BINAIRE	HEXADECIMAL	OCTAL
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Figure 1.8. — Codes hexadécimaux

Exemple : 1010 0001 binaire est représenté par

A 1 en hexadécimal

Exercice 1.25 : *Quelle est la représentation hexadécimale de « 10101010 » ?*

Exercice 1.26 : *Inversement, quel est l'équivalent binaire de l'hexadécimal « FA » ?*

Exercice 1.27 : *Comment s'écrit « 0100 0001 » en octal ?*

L'hexadécimal offre l'avantage de représenter huit bits avec seulement deux symboles. Deux chiffres hexadécimaux sont plus faciles à assimiler, à mémoriser, plus rapides à taper sur un clavier, que leurs équivalents binaires. Par suite, l'hexadécimal est la méthode de prédilection utilisée pour représenter les groupes de bits sur tous les nouveaux micro-ordinateurs.

Naturellement, lorsque l'information de la mémoire a une signification particulière — représentation d'un texte ou d'une série de nombres — l'hexadécimal n'est pas adapté pour, hors de la machine, présenter cette information à l'homme de manière signifiante et utilisable.

Représentation symbolique

La *représentation symbolique* désigne la représentation externe de l'information, sous sa forme symbolique proprement dite.

Par exemple, les nombres décimaux sont présentés comme tels, et non comme une suite de symboles hexadécimaux ou de bits. De même, un texte est présenté en clair. Naturellement, la représentation symbolique est la plus pratique pour l'utilisateur. Elle est utilisée chaque fois qu'un dispositif d'affichage approprié est disponible : imprimante ou console de visualisation (1). Malheureusement, avec de petits systèmes tels que les microordinateurs sur une carte, de tels affichages seraient trop onéreux. La communication entre l'utilisateur et l'ordinateur reste donc limitée à l'hexadécimal.

Résumé des représentations externes.

La représentation symbolique de l'information est la plus souhaitable, parce que la plus naturelle, pour l'utilisateur humain. Elle nécessite cependant un interface coûteux, constitué d'un clavier alphanumérique et d'une imprimante, ou d'un affichage sur tube cathodique. Pour cette raison, la représentation symbolique peut ne pas être proposée sur des systèmes moins sophistiqués et moins onéreux. Un autre type de représentation est alors proposé en remplacement. Dans ce cas, l'hexadécimal prédomine. La représentation binaire n'est utilisée que dans de rares cas, nécessitant une mise au point précise, tant au niveau « hardware » que « software ». Le binaire fait apparaître directement le contenu des cellules mémoires en format binaire. (L'utilité de l'affichage direct en binaire sur le tableau de commande, a toujours fait l'objet d'un débat passionné dans lequel nous ne rentrerons pas ici.)

(1) Une console de visualisation est un système à tube cathodique, semblable à un écran de télévision, servant à l'affichage d'un texte ou de schémas.

Nous venons d'examiner la manière de représenter l'information, tant de manière interne qu'externe. Nous nous intéresserons maintenant au microprocesseur réel, grâce auquel nous manipulerons cette information.

Exercices complémentaires :

Exercice 1.28 : *Quel est l'avantage du complément à deux sur les autres représentations utilisées pour représenter les nombres signés ?*

Exercice 1.29 : *Comment représenteriez-vous 1024 en binaire direct ? en binaire signé ? en complément à deux ?*

Exercice 1.30 : *Qu'est-ce que le bit V. Le programme doit-il le tester après une addition ou une soustraction ?*

Exercice 1.31 : *Calculez le complément à deux de + 16, + 17, + 18, − 16, − 17, − 18.*

Exercice 1.32 : *Représentez, en hexadécimal, quand il est rangé en mémoire en format ASCII sans parité, le texte suivant ? : « MESSAGE ».*

2

L'ORGANISATION HARDWARE DU Z80

INTRODUCTION

Programmer à un niveau élémentaire ne nécessite pas une compréhension détaillée de la structure interne du processeur utilisé. Mais pour programmer de manière efficace, il est, par contre, indispensable d'acquérir cette compréhension. Ce chapitre a pour but de présenter les concepts hardware fondamentaux qui vous permettront de saisir le fonctionnement du système Z80. Un système microordinateur complet ne se limite pas au microprocesseur (ici le Z80). Il comprend aussi d'autres composants. Ce chapitre ne traite que du Z80 lui-même, tandis que les autres composants (principalement des circuits d'entrées/sorties) seront abordés dans un chapitre ultérieur (chapitre 7).

Nous allons à présent aborder les bases de l'architecture d'un microordinateur, avant d'étudier, plus précisément, l'organisation interne du Z80. En particulier, les différents registres, ainsi que le mécanisme de séquencement et d'exécution d'un programme. Ce chapitre ne donne qu'une présentation simplifiée du fonctionnement hardware d'un microprocesseur. Le lecteur désireux d'acquérir une connaissance plus détaillée est invité à se reporter à notre livre réf. C4 (« Les microprocesseurs », du même auteur).

Le Z80 a été conçu comme produit de substitution de l'Intel 8080. Par rapport à ce dernier, il offre toute une gamme de possibilités nouvelles. Nous ferons plusieurs fois référence au 8080, dans ce chapitre.

ARCHITECTURE DU SYSTÈME

L'architecture du microordinateur est présentée à la figure 2.1. Le microprocesseur (MPU : de l'anglais Micro Processor Unit), ici un Z80, apparaît sur la partie gauche de l'illustration. Il englobe en un seul boîtier les fonctions d'une *unité centrale* (CPU : Central Processing Unit) : une *unité arithmétique et logique* et ses registres internes (ALU : Arithmetic-

Logical Unit), et une *unité de commande* (CU : Control Unit), dont la fonction est d'ordonner les opérations du système. Son fonctionnement sera expliqué dans ce chapitre.

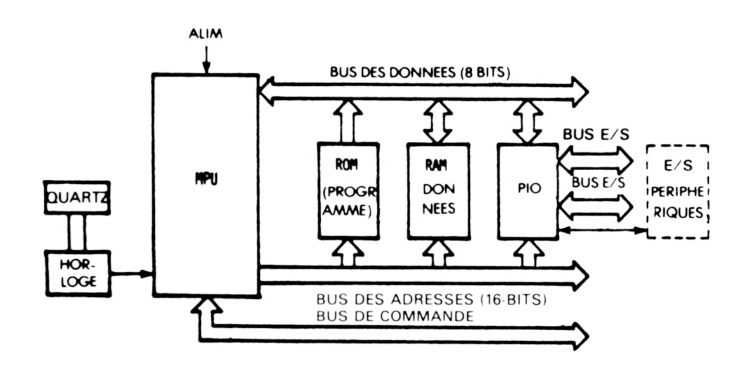


Figure 2.1. — Un système Z80 standard

Le MPU crée trois *bus* : *un bus de données* bidirectionnel de 8 bits (au sommet de la figure), *un bus d'adresses* unidirectionnel de 16 bits et *un bus de commande,* qui apparaît au bas de l'illustration. Décrivons la fonction de chacun de ces bus.

Le *bus de données* transporte les données échangées entre les différents éléments du système. Le transfert a lieu, par exemple, de la mémoire vers le MPU, du MPU vers la mémoire ou encore du MPU vers un boîtier d'entrées/sorties (1)

Le bus d'adresses transporte les adresses générées par le MPU, pour sélectionner un registre interne de l'un des boîtiers connectés au système. Cette adresse précise la source, ou la destination, des données qui transiteront sur le bus de données.

Le *bus de commande* transporte les divers signaux de synchronisation nécessaires au fonctionnement du système.

Le rôle des bus étant connu, connectons les composants supplémentaires indispensables à la formation d'un système complet.

Tout MPU requiert une base de temps précise, fournie par une *horloge* et un *quartz.* Dans les premiers microprocesseurs, l'oscillateur de l'horloge était extérieur au MPU, et nécessitait un boîtier supplémentaire. Aujourd'hui, cet oscillateur est le plus souvent incorporé, bien que tel ne soit pas le cas du Z80. Le quartz, cependant, est toujours extérieur au système, en raison de sa taille. Le quartz et l'horloge apparaissent à gauche de la case MPU sur la figure 2.1.

(1) Un boîtier d'entrées/sorties est un composant permettant les communications entre le MPU et un équipement extérieur.

Portons maintenant notre attention sur les autres composants du système. De gauche à droite, sur l'illustration, nous distinguons :

La ROM (de l'anglais Read Only Memory), mémoire sur laquelle seules les opérations de lecture sont possibles, contient le *programme*. L'avantage d'une mémoire ROM tient au fait que son contenu est permanent : il ne disparaît pas lorsque l'on débranche le système. Par suite, la ROM contient toujours un *programme d'initialisation* (en anglais : bootstrap) ou *de contrôle* (en anglais : monitor), dont le rôle sera précisé plus loin, et qui permet d'initialiser le fonctionnement du système. Dans un contexte de contrôle de processus, presque tous les programmes résideront en ROM, car ils ne seront presque jamais modifiés. L'utilisateur industriel doit en effet se protéger contre les pannes de courant : les programmes ne doivent pas s'effacer. Leur lieu de résidence est en ROM.

Au contraire, l'amateur, ou le programmeur qui développe un programme en vue de le tester, feront résider les programmes en RAM, de manière à pouvoir les modifier facilement. Après la phase de mise au point, ils pourront soit demeurer en RAM, soit, si on le désire, être transférés sur ROM. Il faut savoir que la RAM est volatile, et que son contenu est perdu chaque fois que l'on éteint le système.

La RAM (de l'anglais Random-Access Memory, mémoire à accès aléatoire) est une mémoire que l'on peut lire, et sur laquelle on peut aussi écrire. Dans un système de contrôle, la quantité de RAM sera généralement réduite à la taille nécessaire au rangement des données.

Au contraire, dans un contexte de développement de programmes, la quantité de RAM sera plus importante, pour contenir à la fois les programmes et les données. Il est indispensable de charger dans la RAM un contenu initial avant de pouvoir l'utiliser.

Le système comprend un ou plusieurs boîtiers d'interface, de manière à pouvoir communiquer avec le monde extérieur. Le boîtier d'interface le plus couramment utilisé est le boîtier d'interface parallèle PI0 (de l'anglais : Parallel Input-Output). On peut voir sur l'illustration que le PI0, de même que tous les autres boîtiers du système, est connecté aux trois bus : il fournit au moins deux portes de 8 bits chacune pour la communication avec le monde extérieur. Le lecteur est invité à se reporter au livre C4, pour plus de détails sur le fonctionnement d'une interface parallèle d'entrée-sortie, et au chapitre 7 pour des compléments sur les circuits d'interface du système Z80.

Tous les modules fonctionnels précédemment décrits ne se trouvent pas nécessairement sur un boîtier LSI différent. Il existe des *boîtiers combinés* capables, par exemple, d'inclure à la fois une PIO et un peu de mémoire ROM ou RAM.

D'autres composants entrent en ligne de compte dans la construction d'un véritable système. Ainsi, les bus ont-ils généralement besoin de la présence de circuits « relais » (en anglais « buffers »), capables d'assurer correctement la propagation de leurs signaux à travers le système.

Une *logique de décodage* pourra être utilisée pour les boîtiers de mémoire RAM. Enfin, quelques signaux devront être amplifiés et mis en forme par des circuits nommés « drivers ». Ces circuits auxiliaires ne seront pas décrits ici, car ils n'influencent en rien la programmation. Le lecteur intéressé par

les techniques d'assemblage et d'interface trouvera profit à la lecture de notre livre C5 (« Techniques d'interface des microprocesseurs »).

A L'INTÉRIEUR D'UN MICROPROCESSEUR

La plupart des microprocesseurs disponibles sur le marché présentent la même architecture. Nous nous attacherons à décrire cette architecture « standard », représentée ici à la figure 2.2. Les modules constituant ce microprocesseur seront ensuite détaillés, un par un et de droite à gauche.

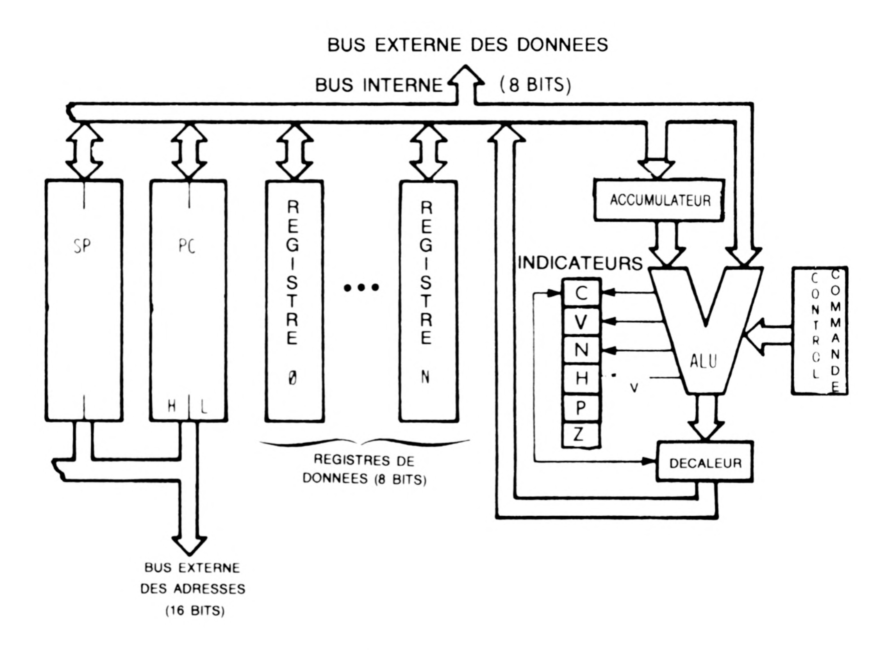


Figure 2.2. — L'architecture « standard » d'un microprocesseur

La case *commande,* sur la droite, représente l'unité de commande synchronisant l'ensemble du système. Son rôle sera précisé au cours de ce chapitre.

L'ALU accomplit les opérations arithmétiques et logiques. Elle est dotée, à l'une de ses entrées (ici, la gauche), d'un registre spécial appelé accumulateur (on peut parfois trouver plusieurs accumulateurs). L'accumulateur présente la particularité de pouvoir être désigné, dans la même instruction, à la fois comme opérande source et comme opérande résultat.

L'ALU doit aussi pouvoir réaliser les opérations de *décalage* et de *rotation.*

Un décalage consiste à déplacer le contenu d'un octet d'une ou plusieurs positions vers la gauche ou vers la droite. Cette opération est illustrée sur la figure 2.3, où chaque bit a été déplacé d'une position vers la gauche. Le détail des opérations de décalage et de rotation sera présenté au prochain chapitre.

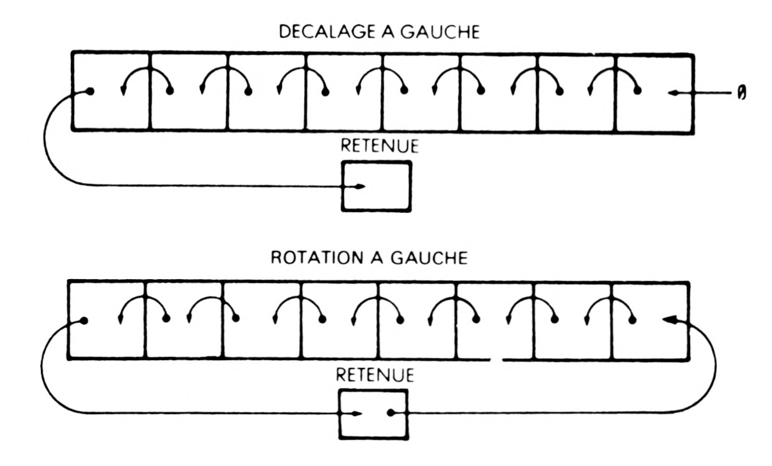


Figure 2.3. — Décalage et rotation

L'opérateur de décalage peut être situé à la sortie de l'ALU (comme indiqué sur la figure 2.2), ou à l'entrée de l'accumulateur.

A gauche de l'ALU, se trouvent les *indicateurs* ou *registre d'état.* Leur rôle est de mémoriser les situations exceptionnelles qui peuvent survenir pendant le fonctionnement du microprocesseur. Le contenu du registre d'état peut être testé par des instructions spécialisées, ou lu sur le bus de données interne. Une *instruction conditionnelle* provoquera, ou ne provoquera pas, l'exécution d'une nouvelle partie du programme, selon la valeur de l'un de ces bits.

Le rôle des indicateurs dans le Z80 sera présenté plus loin dans ce chapitre.

Positionnement des indicateurs

La plupart des instructions exécutées par le processeur modifient tout ou partie des indicateurs. Il est important de toujours se référer à la carte de programmation fournie par le constructeur. Cette carte spécifie les bits appelés à être modifiés par les instructions. Il est essentiel de connaître le fonctionnement des indicateurs pour comprendre le mode d'exécution d'un programme. La figure 4.17 montre le fonctionnement des indicateurs du Z80.

Les registres

Examinons maintenant la figure 2.2. Sur la gauche, on remarque les registres du microprocesseur : *registres à usage universel* et *registres d'adresses.*

Les registres à usage universel

Le rôle des registres à usage universel est de permettre à l'ALU de manipuler les données à grande vitesse. En raison du nombre restreint de bits qu'il est raisonnable d'allouer à chaque instruction, le nombre de registres (directement adressables) est généralement inférieur à 8. Chaque registre est un ensemble de 8 bascules, connectées au bus de données bidirectionnel interne. Ces 8 bits peuvent être simultanément chargés à partir du contenu du bus, ou recopiés sur celui-ci. Leur implémentation en technologie MOS constitue le niveau de mémoire le plus rapide. On peut accéder à leur contenu en quelques dizaines de nanosecondes.

Les registres internes sont généralement numérotés de 0 à n. Leur rôle n'est pas défini à l'avance. C'est pourquoi on les appelle « registres à usage universel ». Ils peuvent contenir n'importe quelle donnée utilisée par le programme.

Ces registres sont habituellement utilisés pour ranger des données de 8 bits. Sur certains microprocesseurs (dont le Z80), il est possible de manipuler deux registres à la fois. On les appelle alors « paires de registres ». Ces paires facilitent les manipulations de quantités sur 16 bits, qu'il s'agisse de données ou d'adresses.

Les registres d'adresses

Ce sont des registres de 16 bits, souvent nommés *pointeurs,* prévus pour contenir des adresses. Leur caractéristique essentielle est d'être connectés au bus d'adresses. Les registres d'adresses émettent les signaux transitant sur le bus d'adresses. Ce dernier apparaît à gauche et en bas de la figure 2.4.

La seule manière de charger le contenu de ces registres de 16 bits est d'utiliser le bus de données, qui ne possède, pour sa part, que 8 bits. Deux transferts sont donc nécessaires. Pour bien distinguer la partie basse (bits de faibles poids) et la partie haute (bits de poids forts) de chaque registre, on adjoint généralement un indice à la partie basse (bits de 0 à 7), au moyen de la lettre « L » (de l'anglais : low), et à la partie haute (bits 8 à 15) au moyen de la lettre « H » (de l'anglais : high). Ces indices sont utilisés chaque fois qu'il est nécessaire de distinguer les deux moitiés de ces registres. Chaque microprocesseur compte au moins deux registres d'adresses. Le mot « MUX », dans la figure 2.4, désigne un multiplexeur.

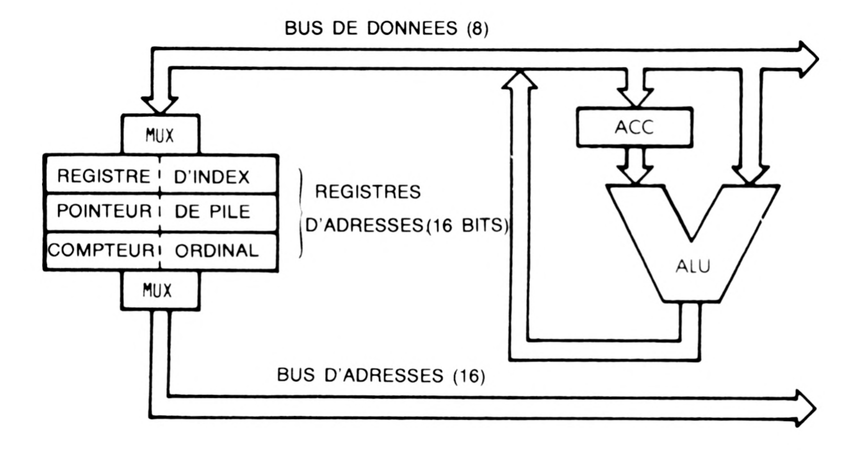


Figure 2.4. — Les registres d'adresses (16 bits) alimentent le bus d'adresses

Le compteur Ordinal (PC)

Le Compteur Ordinal (en anglais : PC, Program Counter) est présent dans tout microprocesseur : il contient l'adresse de la prochaine instruction à exécuter. Cette présence est indispensable et fondamentale pour l'exécution des programmes. Le mécanisme de l'exécution d'un programme, et le séquencement automatique opéré par le compteur ordinal seront décrits dans le prochain paragraphe. Pour l'instant, nous nous contenterons d'indiquer que l'exécution d'un programme se fait, normalement, de façon séquentielle. Pour accéder à l'instruction suivante, il est nécessaire de la transférer de la mémoire vers le microprocesseur. Le contenu du PC est déposé sur le bus d'adresses et transmis à la mémoire, qui lit la valeur présente à l'adresse spécifiée, et la transfère, vers le MPU, sur le bus de données. Cette valeur représente l'instruction recherchée.

Quelques rares microprocesseurs, tels le F8 en deux boîtiers, ne comportent pas de PC dans le MPU. Cela ne signifie pas que le système ne dispose pas d'un compteur ordinal. Le PC est alors, pour des raisons d'efficacité, situé directement sur le boîtier mémoire.

Le pointeur de pile (SP)

Nous n'avons pas encore parlé de *pile,* nous réservant pour un prochain paragraphe. Disons simplement que, dans la plupart des microprocesseurs puissants à usage général, la pile est réalisée par « software », c'est-à-dire dans la mémoire. Pour toujours connaître l'emplacement, dans la mémoire, du sommet de cette pile, un registre spécial lui est affecté : le *pointeur de pile* (SP, de l'anglais : Stack Pointer). Le registre SP contient l'adresse mémoire du sommet de la pile. Nous montrerons que la pile est indispensable au fonctionnement des sous-programmes et des interruptions.

Le registre d'Index (IX)

L'indexation est une technique d'adressage de la mémoire qui n'existe pas sur tous les microprocesseurs. Les diverses techniques d'adressage mémoire seront décrites au chapitre 5. L'indexation permet d'accéder facilement, au moyen d'une seule instruction, aux éléments successifs d'un bloc de données en mémoire.

Généralement, un *registre d'index* contient un déplacement qui sera automatiquement ajouté à une base pour obtenir l'adresse désirée. Il peut aussi contenir une base qui sera ajoutée à un déplacement. En bref, l'indexation est utilisée lorsqu'il s'agit d'accéder à un mot situé dans un bloc de données.

La pile

Formellement, *une pile* est une structure LIFO (de l'anglais : Last In-First Out, dernier entré-premier sorti). Plus précisément, elle est constituée d'un ensemble de registres, ou d'emplacements mémoire, alloué à une telle structure de données. La caractéristique essentielle de cette structure est d'être *chronologique*. Le premier élément introduit dans la pile se trouve toujours au fond de la pile. Le dernier élément déposé se trouve à son sommet. Une analogie peut être faite avec la pile des plateaux situés au début du comptoir d'un self-service. Un ressort est placé de telle sorte qu'il maintienne toujours le plateau le plus haut à portée de main. Les plateaux venant de la plonge sont, au fur et à mesure, déposés sur la pile. C'est toujours le plateau du haut qui est accessible au consommateur. De même, le premier plateau posé est assuré de rester toujours au fond. Autre caractéristique de la pile : elle sera normalement accessible au moyen des seules instructions « empiler » et « dépiler ».

L'*empilage* (en anglais : push) permet de déposer un élément au sommet de la pile (deux dans le cas du Z80). Le *dépilage* (en anglais pull ou pop) consiste à retirer de la pile l'élément qui se trouve à son sommet. Dans le cas d'un microprocesseur, c'est le contenu de *l'accumulateur* qui est déposé au sommet de la pile. Le *dépilage* consiste à transférer le sommet de la pile vers l'accumulateur. D'autres instructions spécialisées peuvent permettre un échange entre le sommet de la pile et d'autres registres, comme le registre d'état. Le Z80 est plus souple et plus puissant que bien des microprocesseurs dans le maniement de la pile.

La présence de la pile est nécessaire à l'implémentation de trois aspects de la programmation d'un ordinateur : les sous-programmes, les interruptions et le stockage temporaire des données. Le rôle de la pile pendant l'exécution des sous-programmes sera expliqué au chapitre 3 (techniques de base de programmation). Son rôle lors des interruptions sera précisé au chapitre 6 (techniques d'entrées-sorties). Enfin, son usage dans la sauvegarde des données à grande vitesse sera présenté dans des programmes d'application spécifiques.

Pour l'instant, nous nous contenterons de supposer que la pile est nécessaire dans tout ordinateur. Elle peut être réalisée de deux façons :

1. Un nombre fixe de registres lui est affecté dans le microprocesseur lui-

même. C'est alors une « pile hardware ». Elle présente l'avantage d'une grande vitesse, mais l'inconvénient d'un nombre limité de registres.

2. De nombreux microprocesseurs ont recours à une autre approche, la pile software, qui permet de supprimer cet inconvénient. C'est l'approche utilisée par le Z80. Un registre est réservé, dans le microprocesseur, pour contenir l'adresse de l'élément situé au sommet de la pile (ou parfois l'adresse à laquelle on pourra empiler le prochain élément). Ce registre spécialisé est le registre SP. La pile est implantée dans une zone de la mémoire. Le pointeur de pile doit donc être de 16 bits pour pouvoir contenir n'importe quelle adresse mémoire.

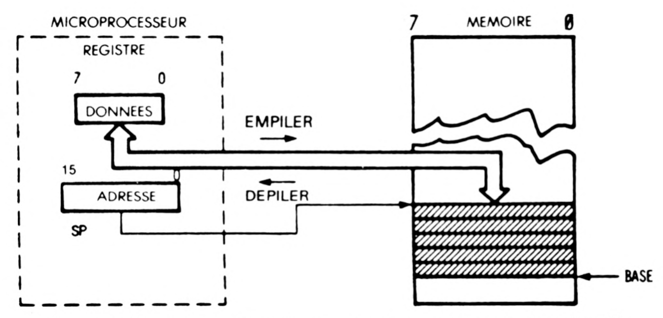


Figure 2.5. — Les deux instructions de manipulation de pile

Le cycle d'exécution d'une instruction

Reportons-nous maintenant à la figure 2.6. Le microprocesseur est représenté à gauche, et la mémoire à droite. Le boîtier mémoire peut être une ROM ou une RAM. Et même, en fait, tout boîtier contenant de la mémoire. La mémoire contient les instructions et les données. Nous allons démontrer le fonctionnement du compteur ordinal en allant chercher une instruction dans la mémoire. Nous supposerons que le compteur ordinal a un contenu valide, qui est l'adresse, sur 16 bits, de la prochaine instruction à exécuter. Quel que soit le processeur, le traitement d'une instruction se fait en trois cycles :

1. récupération de la prochaine instruction
2. décodage de l'instruction
3. exécution de l'instruction

Récupération de l'instruction

Suivons maintenant l'ordre des opérations. Au cours du premier cycle, le contenu du compteur ordinal est déposé sur le bus d'adresses, et transféré vers la mémoire (sur le bus d'adresses). Simultanément, un signal de lecture peut être émis, si nécessaire, sur le bus de commande du système. La mémoire reçoit l'adresse. Quelques centaines de nanosecondes plus tard, la mémoire dépose sur le bus de données les 8 bits de données contenus à l'adresse envoyée. Ce mot de 8 bits est l'instruction que nous voulons

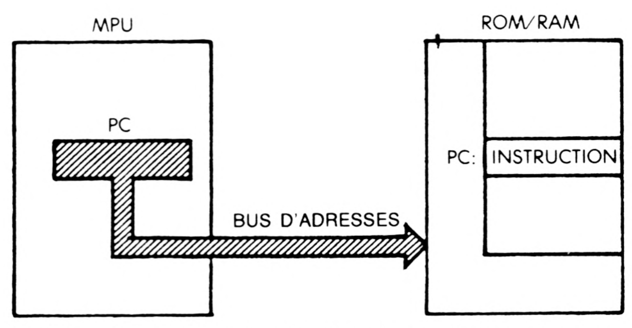


Figure 2.6. — La recherche d'une instruction en mémoire

récupérer. Sur notre dessin, cette instruction sera déposée en haut de la case MPU.

Résumons brièvement la séquence : le contenu du compteur ordinal est envoyé sur le bus d'adresses. Un signal de lecture est généré. La mémoire effectue un cycle. Quelques 300 nanosecondes plus tard, l'instruction qui se trouve à l'adresse spécifiée est déposée sur le bus de données (en supposant qu'il s'agisse d'une instruction d'un seul octet). Le microprocesseur lit alors le bus de données, et dépose son contenu dans un registre interne spécialisé, le registre-instruction IR (de l'anglais : Instruction Register). IR est un registre de 8 bits, dont le rôle est de stocker l'instruction que le MPU vient d'aller chercher en mémoire. Le cycle de recherche est maintenant terminé. Les 8 bits de l'instruction se trouvent physiquement dans le registre du MPU prévu à cet effet : le registre IR. Sur la figure 2.7, ce registre apparaît en haut à gauche. Le registre IR n'est pas accessible au programmeur.

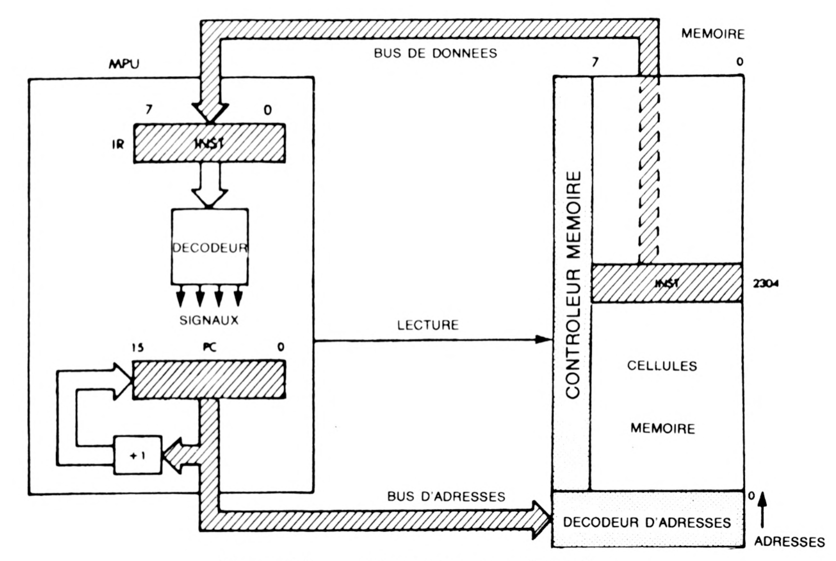


Figure 2.7. — Séquencement automatique

Décodage et exécution

Lorsque l'instruction est parvenue dans le registre IR, l'unité de commande du microprocesseur va en décoder le contenu, de façon à générer la séquence correcte de signaux internes et externes qui permettra l'exécution de l'instruction. Après un court laps de temps consacré au décodage, le microprocesseur va passer à la phase d'exécution, dont la durée dépend de l'instruction considérée. Quelques instructions sont entièrement exécutées à l'intérieur du MPU. D'autres ont besoin d'aller chercher ou de déposer des données en mémoire. C'est la raison pour laquelle les diverses instructions ont des durées d'exécution différentes. La durée d'exécution s'exprime en nombre de cycles (d'horloge). On trouvera au chapitre 4, dans la description précise des instructions, le nombre de cycles requis par chacune. Il est possible, pour un même MPU, d'utiliser plusieurs fréquences d'horloges. On s'explique ainsi que la vitesse d'exécution soit plus volontiers exprimée en nombre de cycles qu'en nombre de nanosecondes.

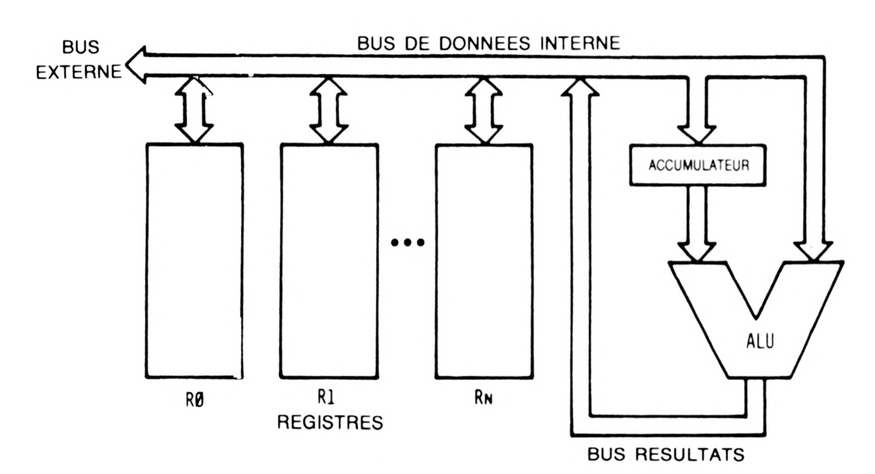


Figure 2.8. — Architecture à un seul bus

Récupération de la prochaine instruction

Nous avons montré de quelle manière, à l'aide du compteur ordinal, il est possible d'aller chercher une instruction en mémoire. Au cours de l'exécution d'un programme, on va chercher les instructions en *séquence,* à l'aide d'un mécanisme automatique. Il s'agit d'un simple incrémenteur, relié au compteur ordinal, que l'on distingue dans la partie inférieure de la figure 2.7. Chaque fois que le compteur ordinal dépose une adresse sur le bus, l'incrémenteur la réécrit, augmentée de 1, dans le compteur ordinal. Exemple : si le compteur ordinal contenait la valeur « 0 », cette dernière

serait déposée sur le bus d'adresses, et la valeur « 1 » mise dans le compteur ordinal. L'instruction située à l'adresse « 1 » serait ainsi recherchée, lors de l'utilisation suivante du compteur ordinal. Cette opération permet d'implémenter un *mécanisme automatique pour la recherche séquentielle des instructions.*

Il est important de souligner que les descriptions données ci-dessus sont simplifiées. En réalité, plusieurs instructions sont composées de deux, ou même trois octets, ce qui implique qu'on aille les chercher successivement en mémoire. Mais le mécanisme reste le même. Le compteur ordinal sert aussi bien à aller chercher les octets successifs d'une même instruction, que les instructions successives elles-mêmes. L'ensemble formé par le compteur ordinal et son incrémenteur constitue un moyen automatique d'adressage d'emplacements mémoires successifs.

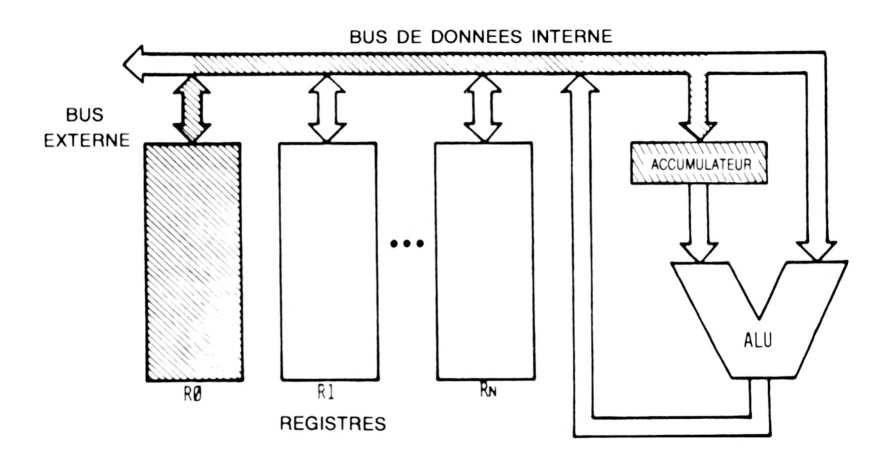


Figure 2.9. — Exécution d'une addition : transfert de R0 dans l'accumulateur.

Nous allons maintenant exécuter une instruction au sein du MPU (voir figure 2.8). Une instruction caractéristique pourrait être, par exemple : R0 = R0 + R1. Ce qui signifie : « additionnez les contenus de R0 et de R1, et rangez le résultat dans R0 ». Cette opération se décompose en plusieurs étapes. Le contenu de R0 est d'abord lu, puis transmis sur le bus unique, déposé à l'entrée de gauche de l'ALU et enfin stocké là, dans le registre tampon qui s'y trouve. R1 est ensuite sélecté, son contenu est lu et déposé sur le bus, puis transféré vers l'entrée de droite de l'ALU.

Cette séquence est illustrée par les figures 2.9 et 2.10. A ce point, les valeurs déposées aux entrées gauche et droite de l'ALU sont, respectivement, celles des registres R0, qui a été copié dans l'accumulateur, et R1, avant le début de l'instruction. L'opération peut avoir lieu. L'addition est effectuée par l'ALU, et le résultat apparaît à la sortie de cette dernière, dans le coin inférieur droit de la figure 2.11. Le résultat est déposé sur le bus

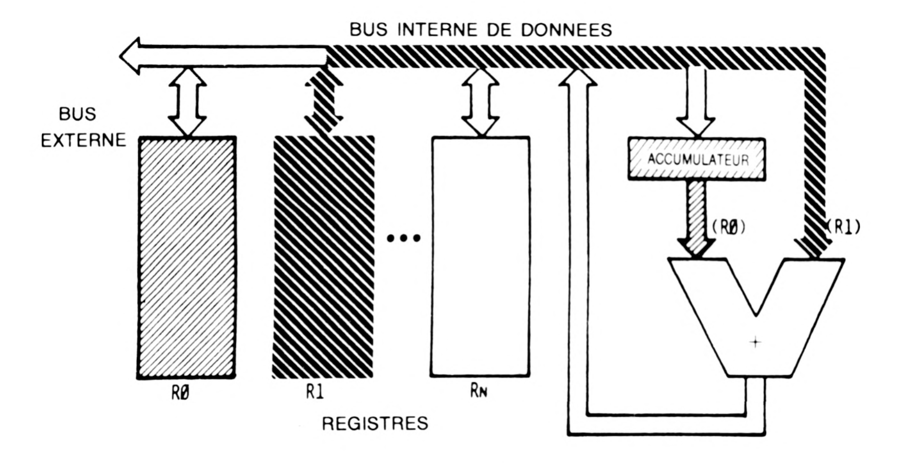


Figure 2.10. — Addition : transfert du second registre dans l'ALU

unique, et transféré vers R0. Concrètement, cela veut dire que le verrou d'entrée de R0 est ouvert, pour que des données puissent y être écrites. Le résultat de l'opération est maintenant dans R0. Il est important de remarquer que cette opération n'a pas modifié le contenu de R1. C'est là un principe général : le contenu d'un registre, ou d'un quelconque emplacement de mémoire RAM, n'est pas modifié par une opération de lecture.

La nécessité du registre tampon à l'entrée gauche de l'ALU est ainsi démontrée. Il était nécessaire que le contenu de R0 soit mémorisé pour que le bus interne unique puisse être utilisé pour effectuer un nouveau transfert, celui de R1. Un problème reste, toutefois, en suspens.

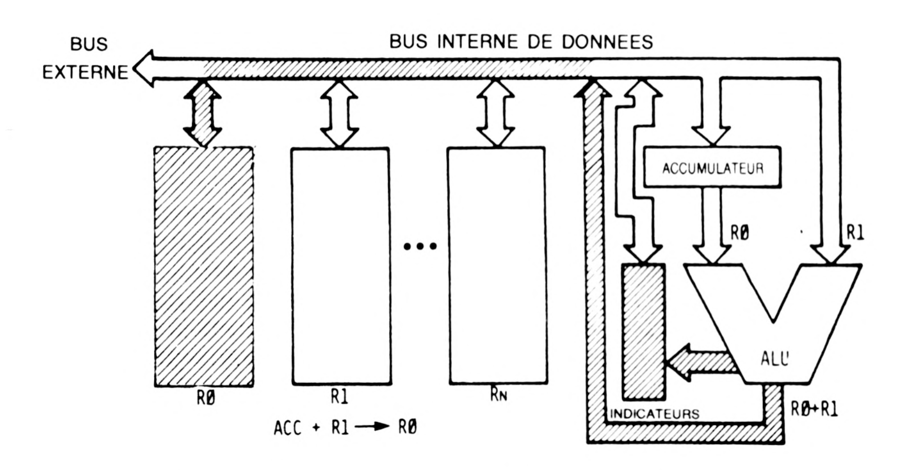


Figure 2.11. — Le résultat est obtenu et transféré vers R0

Le problème de la course-poursuite

L'organisation simplifiée présentée sur la figure 2.8. ne fonctionne pas correctement.

Question : *Un problème de chronologie se pose. Quel est-il ?*

Réponse : Le résultat fourni par l'ALU est déposé sur le bus unique. Il ne va pas se diriger uniquement vers R0, mais aussi se propager sur tout le bus. Il va, en particulier, changer la valeur de l'entrée de droite de l'ALU qui, quelques nanosecondes plus tard, fournira à son tour un nouveau résultat, lequel risque de se substituer au premier, avant même que le dépôt de ce dernier dans R0 soit achevé. Il se produit une *course-poursuite* entre l'entrée de droite de l'ALU et sa sortie. Pour l'éviter, il faudra isoler l'entrée de l'ALU de sa sortie (voir figure 2.12.)

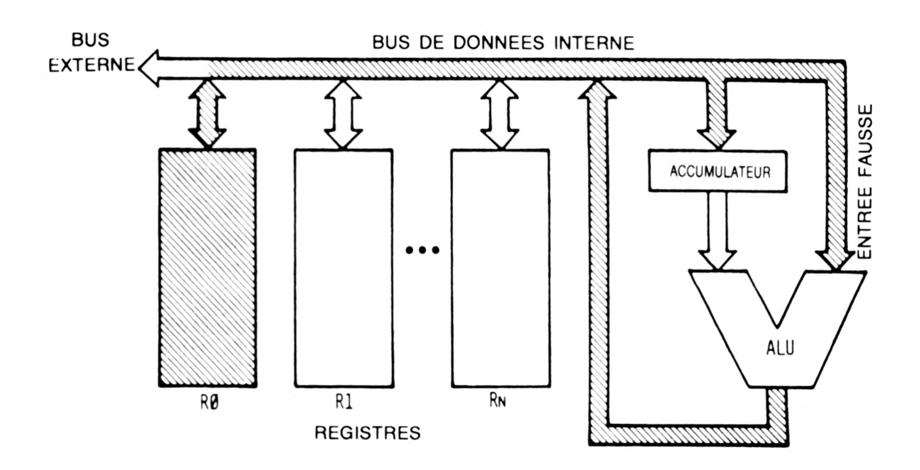


Figure 2.12. — Le problème de la course-poursuite

Pour ce faire, plusieurs solutions sont possibles. On peut avoir recours à un registre tampon placé, soit à la sortie de l'ALU, soit, plus fréquemment, à son entrée. Dans notre cas, c'est sur l'entrée de droite de l'ALU qu'il conviendra de le placer. Le nombre de tampons introduits dans le système est maintenant suffisant pour en assurer le fonctionnement correct. Nous verrons plus loin dans ce chapitre, que si le registre de l'entrée de gauche de l'ALU (visible sur la figure) est utilisé en tant qu'accumulateur (ce qui permet l'utilisation d'instructions d'un seul octet), alors, il sera également nécessaire de placer un tampon entre l'ALU et l'accumulateur, comme présenté sur la figure 2.13.

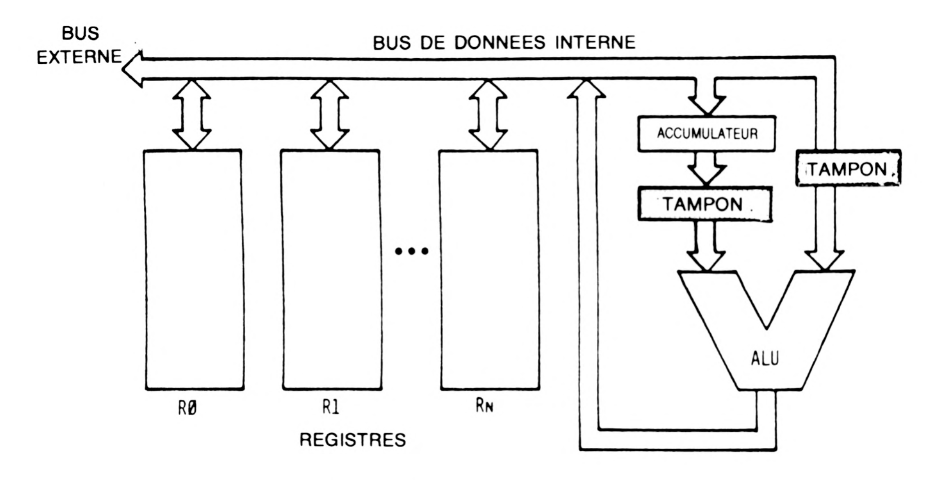


Figure 2.13. — Deux tampons sont nécessaires

L'ORGANISATION INTERNE DU Z80

Tous les termes nécessaires à la compréhension des éléments internes du microprocesseur ont maintenant été définis, ce qui va nous permettre d'examiner plus en détail le Z80 lui-même, et d'exposer ses possibilités. La figure 2.14 en présente une description logique, mettant en évidence son organisation interne. D'autres connexions peuvent exister, qui ne seront pas montrées ici. Examinons ce diagramme, de droite à gauche.

Sur la droite se trouve *l'unité arithmétique et logique* (l'ALU) reconnaissable à sa forme caractéristique en « V ». L'accumulateur, décrit dans la section précédente, se signale par la lettre A, sur l'entrée de droite de l'ALU. Nous avons vu, dans le paragraphe précédent, qu'il est nécessaire de mettre un registre tampon entre l'accumulateur et l'entrée de l'ALU. C'est ici le registre marqué ACT (Accumulateur Temporaire). L'entrée gauche de l'ALU est aussi munie d'un *registre temporaire*, TMP. Le fonctionnement de l'ALU sera détaillé dans le prochain paragraphe, avec la description des instructions du Z80.

Le *registre d'indicateurs* du Z80 s'appelle « F » (de l'anglais : Flags, indicateurs) ; il est dessiné à droite de l'accumulateur. La valeur des indicateurs dépend essentiellement de l'ALU, mais nous verrons que certains indicateurs peuvent dépendre d'autres modules ou événements.

L'accumulateur et le registre d'indicateurs sont présentés sous forme de doubles registres : A, A′ et F, F′.

En fait, le Z80 est équipé de deux couples de registres : A + F, et A′ + F′. *Un seul* de ces couples est accessible à un moment donné. Une instruction spéciale permet d'échanger les contenus de A et F et ceux de A′ et F′. Afin de simplifier les explications, seuls A et F seront dessinés sur la plupart des diagrammes qui suivront. Le lecteur se souviendra qu'il a la

possibilité d'utiliser la paire A', F' lorsqu'il le désire, à la place de la paire A, F.

Le rôle de chaque indicateur du registre F sera décrit au chapitre 3 (Techniques de base de programmation).

On remarque au centre du diagramme un grand bloc de registres. Dans sa partie supérieure, se trouvent deux groupes identiques, contenant chacun six registres, appelés B, C, D, E, H et L. Ce sont les *registres 8 bits universels* du Z80. Le Z80 se distingue du microprocesseur standard étudié au début de ce chapitre par deux particularités.

La première est que le Z80 possède *deux* groupes identiques de 6 registres. Seuls, six de ces registres sont utilisables à un moment donné. Mais des instructions spéciales permettent de choisir le groupe sur lequel on désire travailler. L'utilisateur dispose ainsi de la possibilité d'utiliser un des groupes comme mémoire, et l'autre comme ensemble de registres de travail. Les possibilités offertes par l'existence de ce second groupe seront décrites dans le prochain chapitre.

Pour le moment, afin d'éviter toute confusion, nous considérerons qu'il n'existe que six registres de travail, B, C, D, E, H et L, en ignorant le second groupe.

Le symbole MUX qui apparaît au-dessus des groupes de registres est l'abréviation de *Multiplexeur*. Les données provenant du bus de données interne sont aiguillées par le multiplexeur vers le registre sélectionné. Un seul de ces registres peut, à un moment donné, être connecté au bus de données interne.

La deuxième particularité de ces six registres, en plus d'être des registres 8 bits universels, est d'être connectés au *bus d'adresses*. Ils sont, pour cette raison, associés par *paires*. Les contenus de B et de C peuvent ainsi être dirigés simultanément vers le bus d'adresses de 16 bits qui apparaît en bas de la figure. Ce groupe peut donc recevoir soit des données sur 8 bits, soit des *pointeurs* sur 16 bits à des fins d'adressage mémoire.

Le troisième groupe de registres, qui apparaît sous les deux autres, au milieu de la figure 2.14, contient quatre registres destinés exclusivement à l'adressage. Comme dans tout microprocesseur, nous trouvons le compteur ordinal (PC), et le pointeur de pile (SP). Rappelons que le compteur ordinal contient l'adresse de la prochaine instruction à exécuter.

Le pointeur de pile permet de repérer le sommet de la pile en mémoire. Dans le Z80, il pointe sur la dernière entrée de la pile occupée. (Dans certains microprocesseurs, le pointeur de pile pointe juste au-dessus du sommet de la pile, c'est-à-dire à l'emplacement qui sera occupé par la prochaine donnée empilée). Dans le Z80, la pile *« monte »* vers les adresses *basses* de la mémoire.

Le pointeur de pile doit donc être *décrémenté* chaque fois qu'un mot nouveau est *empilé* (au moyen d'une instruction « push »). Réciproquement, chaque fois qu'un mot est *enlevé* (dépilé), le pointeur de pile doit être *incrémenté* de 1. Dans le cas du Z80, les instructions « push » et « pop » empilent ou dépilent *deux* mots à la fois. Le pointeur de pile doit donc être décrémenté ou incrémenté de deux.

Les deux autres registres de ce groupe de quatre, nous font découvrir un nouveau type : les *registres d'index*. Ici : IX (registre d'index X) et IY (registre d'index Y). Ces deux registres sont munis d'un additionneur spécial, dessiné sur leur droite, avec la forme en « V » d'une petite ALU. Un octet provenant du bus de données interne peut être ajouté au contenu de l'un ou l'autre de ces registres, pour fournir une adresse sur le bus de données. Notons que le résultat de cette addition ne modifie pas le contenu du registre choisi. L'octet ajouté au registre d'index s'appelle *déplacement*. Des instructions spéciales provoquent l'addition automatique de ce déplacement au contenu des registres IX ou IY, pour générer une adresse. Cette opération, appelée *indexation*, est un moyen commode d'accès à n'importe quel bloc séquentiel de données. Ce mode d'adressage important sera décrit au chapitre 5, consacré aux techniques d'adressage.

Enfin, une boîte spéciale marquée « ± 1 » apparaît en bas et à gauche de ce groupe de registres. Elle permet, éventuellement, d'incrémenter ou de décrémenter automatiquement les registres SP, PC et les paires de registres BC, DE et HL, aussitôt après leur utilisation (par exemple, chaque fois qu'un registre vient de déposer son contenu sur le bus d'adresses).

C'est une commodité essentielle, qui permet de réaliser automatiquement des *boucles* dans les programmes, notamment pour faciliter l'accès à des emplacements mémoire successifs. Nous examinerons cette possibilité dans le prochain paragraphe.

Regardons maintenant la partie gauche de l'illustration. On y découvre les deux registres I et R. Le registre I sert à *repérer l'emplacement de la table des vecteurs d'interruption*. Son rôle sera décrit aux chapitre 6 (techniques d'entrées-sorties), dans le paragraphe consacré aux interruptions. Il n'est utilisé que dans un cas spécial : lorsque l'adressage indirect d'un emplacement mémoire doit être généré en réponse à une interruption. Le registre I sert alors à contenir les bits 8 à 15 de l'adresse indirecte, les bits 0 à 7 étant fournis par le boîtier générateur de l'interruption.

Le registre R est *le registre de rafraîchissement mémoire*. Il sert à rafraîchir, automatiquement, les mémoires dynamiques. Ce registre était traditionnellement situé à l'extérieur du microprocesseur, puisqu'il est directement lié à la présence de mémoires dynamiques. Son incorporation au microprocesseur permet, dans de nombreux cas, de réduire sensiblement le nombre de boîtiers nécessaires à la connexion de mémoires dynamiques. Il ne sera pas utilisé ici pour la programmation, puisqu'il s'agit, essentiellement, d'un dispositif hardware (voir notre livre C5 « Techniques d'interface des microprocesseurs », pour une description détaillée des techniques de rafraîchissement de la mémoire). Il est pourtant parfois possible d'utiliser ce registre en programmation : pour l'obtention de valeurs « pseudo-aléatoires », par exemple.

Examinons enfin la partie située à l'extrême-gauche de la figure 2.14 : là où se trouve l'unité de commande du microprocesseur. En partant du haut, nous trouvons d'abord le *registre d'instruction* IR, qui contient l'instruction à exécuter. Notons que ce registre n'a rien à voir avec les registres I et R étudiés précédemment. L'instruction, qui provient de la mémoire, via le bus de données, se propage le long du bus interne de données. Elle est

finalement déposée dans le registre d'instruction. Sous ce dernier se trouve le *décodeur,* qui envoie des signaux au contrôleur-séquenceur, provoquant l'exécution de l'instruction dans, et à l'extérieur, du microprocesseur. *L'unité de commande* alimente et gère le bus de commande qui apparaît dans la partie inférieure de l'illustration.

Les trois bus gérés ou alimentés par le système, à savoir le bus de données, le bus d'adresses et le bus de commande, se prolongent à l'extérieur du boîtier du microprocesseur, à travers ses broches. Les bus sont isolés de l'extérieur par des tampons, que l'on peut voir sur la figure 2.14.

Tous les éléments de la logique du Z80 ont maintenant été décrits. Il n'est pas essentiel d'en comprendre le détail du fonctionnement interne pour

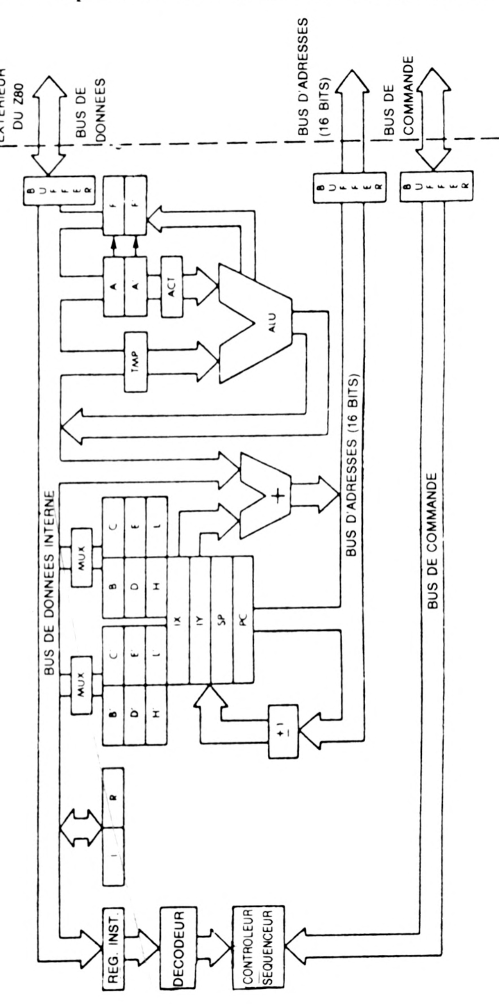


Figure 2.14. — Organisation interne du Z80

entreprendre l'écriture de programmes. Cependant, le programmeur désireux d'écrire des programmes performants doit savoir que la vitesse d'exécution et la taille d'un programme dépendent de la justesse du choix des registres et des techniques. Seule, une bonne compréhension de la façon dont le microprocesseur exécute les instructions permet de faire un choix judicieux. Afin d'illustrer le rôle et l'utilisation des registres internes et des bus, nous examinerons en détail l'exécution de quelques instructions types.

FORMAT DES INSTRUCTIONS

Les instructions du Z80 (listées au Chap. 4) peuvent comporter un, deux, trois ou quatre octets. Une instruction sert à spécifier l'action que doit accomplir le microprocesseur. En simplifiant, on pourrait dire que chaque instruction est constituée d'un code-opération, suivi d'un opérande optionnel, sur un ou deux mots, qui peut être un littéral ou une adresse. Le code-opération précise l'opération à effectuer. D'un strict point de vue terminologique, le code-opération comprend uniquement les bits destinés à préciser cette opération, à l'exclusion de ceux pouvant être, éventuellement, chargés de désigner les registres nécessaires. Dans le monde de la micro-informatique, il est d'usage de regrouper sous le nom de code-opération, non seulement le code-opération lui-même, mais aussi tous les pointeurs de registres que peut contenir l'instruction. A des fins d'efficacité, ce « code-opération généralisé » doit tenir dans un mot de 8 bits. (Cette contrainte limite le nombre d'instructions disponibles sur un microprocesseur).

Le 8080 dispose d'instructions sur un, deux ou trois octets (voir Fig. 2.15). Le Z80 est muni, en outre, d'instructions d'indexation, qui exigent un octet supplémentaire. Dans le cas du Z80, le code-opération tient, en général, sur un octet, sauf pour quelques instructions spéciales, qui en demandent deux.

Certaines instructions nécessitent, derrière le code-opération, la présence d'un octet de données. Dans ce cas, on aura une instruction sur deux octets (sauf pour l'indexation, qui demande un octet supplémentaire).

D'autres demandent qu'une adresse soit fournie. Comme il faut 16 bits, soit 2 octets, pour préciser une adresse, nous aurons dans ce cas une instruction de trois octets, ou même de quatre, lorsque le code-opération comporte déjà deux octets.

Quatre cycles d'horloge sont nécessaires à l'unité de commande pour récupérer, depuis la mémoire, chaque octet du code-opération, et trois cycles pour récupérer les octets suivants de l'instruction. On comprend donc que, généralement, plus courte sera l'instruction, plus rapide sera son exécution.

Une instruction d'un seul mot

Les instructions sur un seul mot sont, en principe, les plus rapides. Ce sont celles que préfère le programmeur. Un exemple-type d'une telle instruction du Z80 est :

LD r, r'

Cette instruction signifie : « Transférez le contenu du registre r' dans le registre r ». Il s'agit, typiquement, d'une opération de registre à registre. Tout microprocesseur doit posséder de telles instructions, qui permettent au programmeur de transférer le contenu de n'importe quel registre dans n'importe quel autre. Les instructions travaillant sur des registres spéciaux de la machine, tels que l'accumulateur, peuvent disposer d'un code-opération spécial.

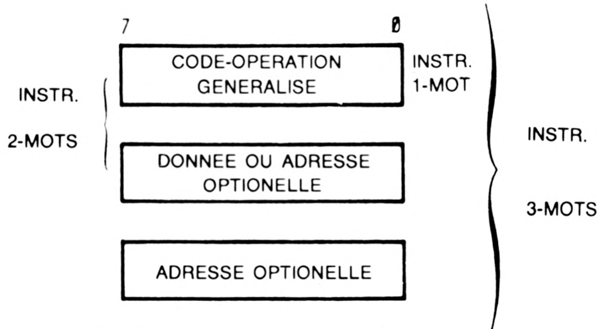


Figure 2.15. — Formats types d'instructions

Après exécution de l'instruction ci-dessus, le contenu du registre r est devenu égal au contenu du registre r'. Ce dernier *n'a pas* été modifié par l'opération de lecture.

Toute instruction doit être représentée, à l'intérieur du microprocesseur, dans un format binaire. La représentation « LD r, r' » est dite symbolique ou *mnémonique*. Elle est la représentation de l'instruction en *langage assembleur*, et permet simplement de disposer d'une représentation symbolique pratique du code binaire réel de cette instruction. Le code binaire représentant cette instruction dans la mémoire est : 01DDDSSS (bits 7 à 0).

Cette représentation reste partiellement symbolique. Chacune des lettres S et D représente un bit. Les trois D, « DDD », sont les 3 bits désignant le registre *destination*. Ils permettent donc la sélection d'un registre, parmi huit possibles. Les codes binaires de ces registres sont représentés sur la Figure 2.16. Par exemple, le code du registre B est « 000 », celui de C, « 001 », etc...

De même, « SSS » représente les 3 bits permettant de désigner le registre *source*. La convention est ici que r' est le registre source, et r le registre destination. La position de ces bits dans la représentation binaire d'une instruction est de peu d'intérêt pour le programmeur, mais présente une grande importance pour l'unité de commande du microprocesseur, qui doit décoder et exécuter l'instruction. En revanche, la représentation de l'instruction en *langage assembleur* est destinée à faciliter la programmation. On pourrait la discuter, en mettant en évidence une interprétation possible : « Transférer le contenu de r dans r' ». En fait, cette convention a été choisie de manière à préserver une compatibilité avec la représentation binaire. Elle est, bien entendu, arbitraire.

Exercice 2.1 : *Écrire le code binaire de l'instruction qui transfère le contenu du registre C dans le registre B. Consulter la figure 2.16. pour obtenir les codes des registres C et B.*

Un autre exemple d'instruction sur un mot est :

ADD A, r

Cette instruction provoque l'addition du contenu du registre spécifié (r) à celui de l'accumulateur (A). Cette opération peut être représentée symboliquement par : A = A + r. On peut vérifier, dans l'appendice E, que la représentation binaire de cette instruction est :

10000SSS

où SSS désigne le registre qu'il convient d'ajouter à l'accumulateur. Les codes des registres sont les mêmes que ceux indiqués dans la figure 2.16.

Exercice 2.2 : *Quel est le code binaire de l'instruction ajoutant le contenu du registre D à celui de l'accumulateur ?*

CODE	REGISTRE
0 0 0	B
0 0 1	C
0 1 0	D
0 1 1	E
1 0 0	H
1-0 1	L
1 1 0	-(MEMOIRE)
1 1 1	A

Figure 2.16. — Les codes des registres

Une instruction de deux mots :

ADD A, n

Cette instruction simple de deux mots additionne le contenu du deuxième octet de l'instruction à celui de l'accumulateur. Le contenu du second octet de l'instruction est appelé un « littéral ». C'est une donnée sur 8 bits qui n'a pas de signification spéciale. Il peut arriver qu'il s'agisse d'un caractère ou d'une valeur numérique, mais cela n'a aucune influence sur l'opération.
Le code de cette instruction est :

11000110 suivi de l'octet « n ».

L'opération est *immédiate*. « Immédiat », dans de nombreux langages de programmation, signifie que le ou les mots suivants de l'instruction contiennent une donnée qui n'a pas à être *interprétée* (comme l'est par

exemple le code-opération). Le mot suivant, ou les deux mots suivants, doivent être traités littéralement, d'où l'appellation *littéral.*

L'unité de commande est programmée pour « savoir » le nombre de mots de chaque instruction. Elle cherchera donc, et exécutera le bon nombre de mots pour chaque instruction. Il faut toutefois savoir que plus le nombre de mots possible pour une instruction est grand, plus le décodage est difficile.

Une instruction de trois mots :

LD A, (nn)

Cette instruction se code sur trois mots et signifie :

« Charger l'accumulateur avec la valeur contenue à l'adresse mémoire précisée dans les deux octets suivants de l'instruction ». Puisque les adresses s'écrivent sur 16 bits, il faudra 2 octets pour coder l'adresse. La représentation binaire de cette instruction est :

00111010	8 bits pour le code-opération
partie basse de l'adresse	8 bits pour la partie basse de l'adresse
partie haute de l'adresse	8 bits pour la partie haute de l'adresse

EXÉCUTION DES INSTRUCTIONS DANS LE Z80

Nous avons vu que toutes les instructions sont exécutées en trois étapes : RÉCUPÉRATION, DÉCODAGE et EXÉCUTION. Nous allons maintenant introduire quelques définitions. L'exécution de chaque étape demande quelques périodes (ou « temps ») d'horloge. Le Z80 exécute chaque étape en un ou plusieurs cycles logiques, appelés « cycles-machine ». Le cycle-machine le plus court est l'équivalent de trois temps d'horloge.

Le cycle-machine de récupération du code-opération d'une instruction correspond à quatre temps d'horloge. L'exécution de l'instruction la plus courte demandera donc au moins quatre temps d'horloge ; la plupart des instructions demandent plus.

Les cycles-machine successifs nécessaires à l'exécution d'une instruction s'appellent M1, M2, etc. Leur exécution demande au moins trois temps d'horloge, appelés T1, T2, etc.

L'étape de RÉCUPÉRATION

Cette étape de l'instruction a lieu pendant les trois premiers temps (T1, T2 et T3) du cycle-machine M1. Ces trois temps sont communs à toutes les instructions du microprocesseur, qui toutes doivent être recherchées en

mémoire avant de pouvoir être exécutées. Le mécanisme de récupération est le suivant :

T1 : ENVOYER PC

La première étape consiste à envoyer à la mémoire l'adresse de la prochaine instruction, située dans le compteur ordinal (PC). Au cours de cette étape, quelle que soit l'instruction à récupérer, le contenu du compteur ordinal est placé sur le bus d'adresses. (cf. Fig. 2.17).

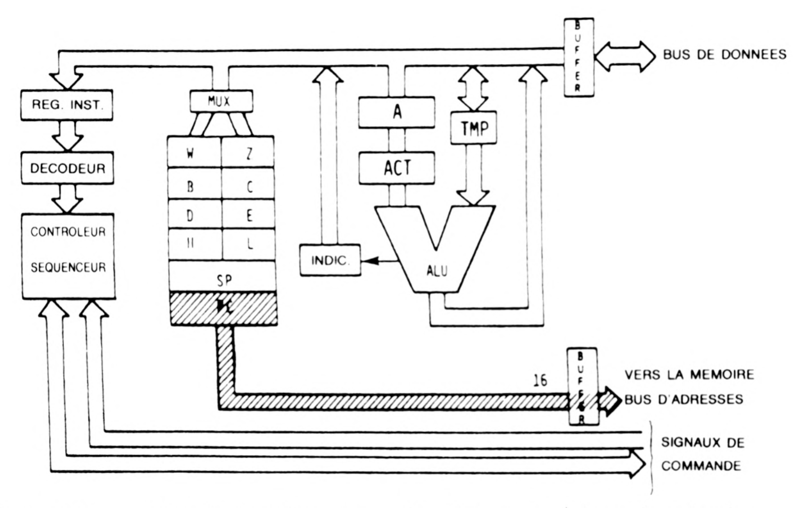


Figure 2.17. — Récupération d'une instruction PC est envoyé vers la mémoire

Lorsqu'une adresse se présente à la mémoire, le décodeur d'adresses va la décoder pour choisir l'emplacement spécifié. Quelques centaines de nanosecondes (1 nanoseconde = 10^{-9} seconde) vont s'écouler avant que le contenu de l'emplacement mémoire sélectionné soit présenté sur les broches de sortie de la mémoire, connectées au bus de données. C'est une caractéristique de tous les ordinateurs de mettre à profit les temps de lecture mémoire pour réaliser quelques tâches. Ici, le travail consistera à incrémenter le compteur ordinal, pendant le temps d'horloge T2 (cf. Fig. 2.18).

T2 : PC = PC + 1

A la fin du temps T2, le contenu de la mémoire est disponible, et peut être transféré à l'intérieur du microprocesseur.

T3 : RANGER L'INSTRUCTION DANS IR.

Dernière phase de récupération de l'instruction. Elle s'achève par le dépôt de cette dernière dans le registre d'instruction IR.

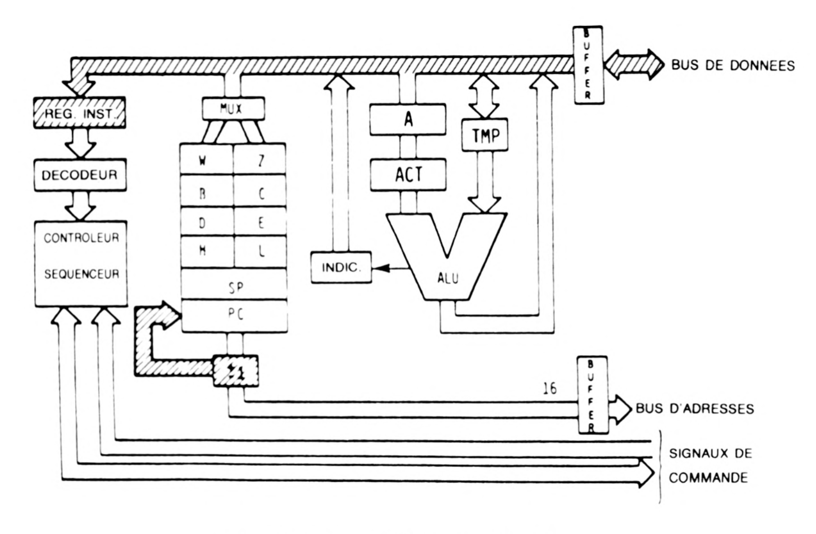


Figure 2.18. — PC est incrémenté

Les étapes de DÉCODAGE et d'EXÉCUTION

Durant le temps T3, l'instruction, qui se trouve maintenant dans le registre IR, va être décodée. Le cycle machine M1 comporte toujours un quatrième temps d'horloge. En effet, une fois l'instruction obtenue, il est indispensable de la *décoder* et de *l'exécuter,* ce qui demande au moins un temps d'horloge supplémentaire, T4.

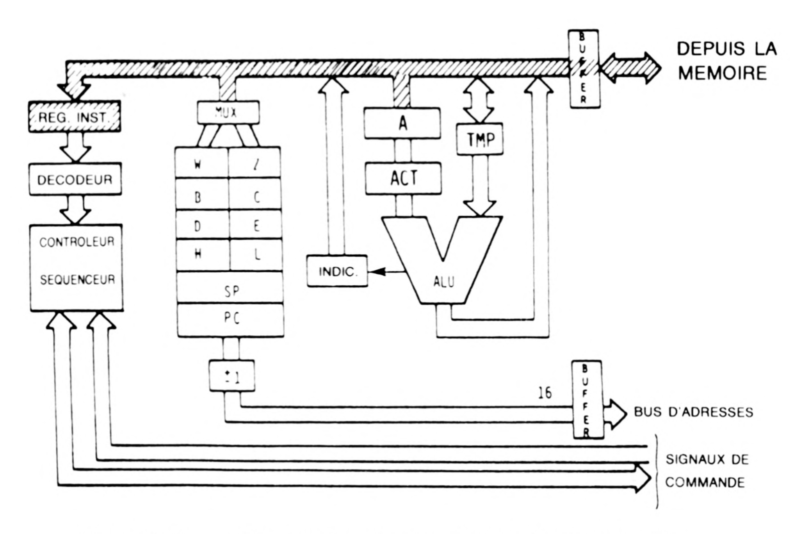


Figure 2.19. — L'instruction chemine de la mémoire vers IR

Quelques rares instructions requièrent un cinquième (T5), voire même un sixième (T6), temps d'horloge pour accomplir le cycle machine M1. Mais, dans la plupart des cas, quatre temps suffisent. Chaque fois que l'exécution de l'instruction demande plusieurs cycles machine (M1, M2, etc.), le temps T1 du cycle M2 suit immédiatement le temps T4 du cycle M1. Etudions un exemple. On trouvera à la figure 2.27 le séquencement interne de chaque instruction. Ces tables ne sont pas publiées pour le Z80. Nous utiliserons, à défaut, celles du 8080, qui permettent une compréhension détaillée de l'exécution d'une instruction.

LD D,C

Cette instruction correspond au MOV r1,r2 du 8080, visible à la ligne 1 de la figure 2.27. Par coïncidence, le registre destination de notre exemple s'appelle « D ». Le transfert du contenu du registre C vers le registre D est illustré figure 2.20.

Cette instruction a été décrite dans un précédent paragraphe. Elle transfère le contenu du registre C dans le registre D.

Les trois premiers temps du cycle M1 sont utilisés à la récupération de l'instruction en mémoire. A la fin de T3, l'instruction se trouve dans IR, où elle peut être décodée (cf. Fig. 2.19).

Pendant T4 : (SSS) → TMP
le contenu de C est déposé dans TMP (cf. Fig. 2.21).

Pendant T5 : (TMP) → DDD
le contenu de TMP est déposé dans D, comme on le voit à la figure 2.22.

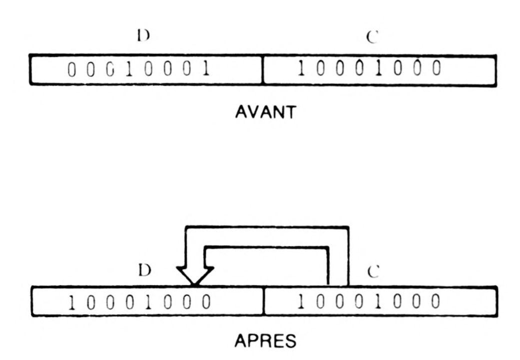


Figure 2.20. — Transfert de C vers D

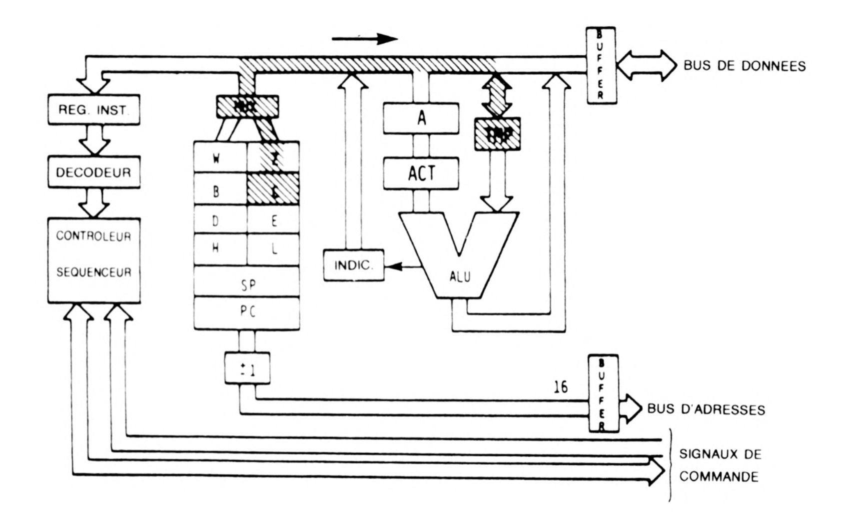


Figure 2.21. — (C) → TMP

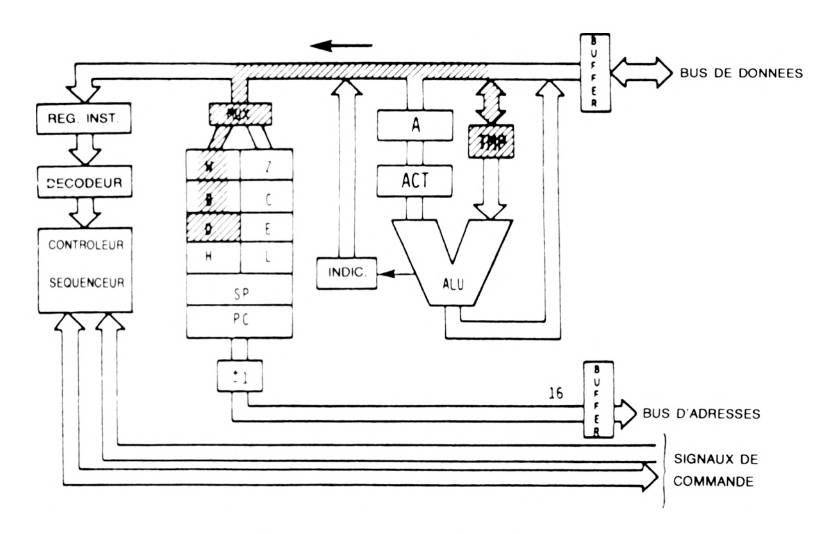


Figure 2.22. — (TMP) → D

Remarque : il est d'usage, en informatique, de désigner le contenu d'un registre ou d'un emplacement mémoire en mettant leurs noms entre parenthèses. Ainsi, s'il advient que le registre C contienne la valeur 4, on aura la relation : (C) = 4.

L'exécution de l'instruction est maintenant terminée. Le contenu du registre C a été transféré dans le registre destination D. Cette instruction ne

nécessite pas d'autres cycles machine, et son exécution s'arrête donc à la fin de M1.

Il est possible de calculer facilement la durée de cette instruction, sachant que la durée de chaque temps, pour le Z80 standard, est la période de l'horloge, soit 400 nanosecondes. La durée de l'instruction, qui demande cinq temps d'horloge, est donc :
$5 \times 400 = 2\,000$ ns = 2 μs (ns et μs sont les abréviations de nanoseconde et microseconde).

Question : *Pourquoi cette instruction demande-t-elle deux temps, T4 et T5, pour transférer le contenu de C vers D, et non pas un seul ? Plutôt que de transférer successivement le contenu de C dans TMP, puis celui de TMP dans D, ne serait-il pas plus simple de transférer directement le contenu de C dans D, en un seul temps ?*

Réponse : Cette solution est impossible, en raison de la manière dont sont construits les registres internes. En fait, tous font partie d'une RAM unique (mémoire lecture/écriture), située dans le boîtier du microprocesseur. On ne peut adresser (sélecter) qu'un seul mot de cette RAM à la fois. Pour lire ou écrire deux mots, deux cycles RAM sont nécessaires. Ainsi, faut-il d'abord lire le contenu du registre C, avant de le déposer dans un registre intermédiaire TMP (qui, lui, ne fait pas partie de la RAM), puis recopier le contenu de TMP dans la RAM. On pensera peut-être qu'il s'agit là d'une erreur de conception du microprocesseur. En fait, cette limitation est commune à presque tous les microprocesseurs composés d'un seul boîtier. Pour résoudre ce problème, il faudrait une RAM disposant de deux portes d'accès simultané. Cette limitation n'est d'ailleurs pas une caractéristique intrinsèque des microprocesseurs, puisque les microprocesseurs en tranches n'en sont normalement pas affectés. Elle est simplement le résultat provisoire d'une recherche constante de densité sur les circuits intégrés, et disparaîtra sans doute dans le futur.

EXERCICE IMPORTANT :

Il est fortement conseillé au lecteur de revoir le séquencement de cette instruction simple, avant que nous n'en étudiions de plus complexes. A cette fin, on pourra se reporter à la figure 2.14, rassembler quelques « symboles » de petite taille (tels qu'allumettes, trombones, etc.) et les déplacer sur la figure pour simuler le cheminement des données, depuis les registres jusqu'aux bus. Par exemple, on déposera un symbole dans PC, que T1 fera sortir pour l'envoyer vers la mémoire, via le bus d'adresses. Il est conseillé, avant d'aborder la suite, de poursuivre cette exécution simulée jusqu'à l'acquisition d'une maîtrise suffisante dans le maniement des transferts entre registres, et le long des bus.

Nous allons maintenant étudier progressivement des instructions plus complexes.

SUB r

Cette instruction signifie : « Soustraire le contenu du registre r (spécifié par un code binaire SSS) de celui de l'accumulateur ». Il s'agit d'une instruction *implicite.* L'appellation implicite tient au fait qu'un seul registre est explicitement nommé (le registre r), alors que l'instruction travaille sur deux registres, le second étant l'accumulateur. Ce dernier, lorsqu'il est ainsi utilisé, est pris à la fois comme opérande source et comme registre destination (lieu de dépôt du résultat). L'avantage d'une instruction implicite est, en général, que son code-opération est susceptible de tenir sur un seul octet. Dans le cas présent, nous n'avons besoin que de trois bits pour désigner le registre impliqué. L'instruction permet d'effectuer rapidement une soustraction.

D'autres instructions implicites existent sur le Z80. Elles portent sur d'autres registres spécialisés. Exemple plus complexe : celui de l'instruction CALL nn, qui met dans le compteur ordinal la valeur nn, après avoir sauvegardé son ancien contenu au sommet de la pile (manipulation implicite du pointeur de pile SP, décrémenté de deux lors de l'empilage de l'ancienne valeur de PC).

Examinons maintenant, en détail, l'exécution de l'instruction SUB r. Instruction qui nécessite deux cycles machine, M1 et M2. Comme d'habitude, durant les trois premiers temps de M1, l'instruction est recherchée en mémoire et déposée dans le registre IR. Au début de T4, elle est décodée, et peut être alors exécutée. Supposons qu'il faille soustraire le contenu du registre B de celui de l'accumulateur. Le code de l'instruction SUB B sera : 10010000 (car le code du registre B est 000). L'équivalent 8080 de cette instruction s'écrit aussi symboliquement SUB r.

T4 : (SSS) → TMP, (A) → ACT

Figure 2.23. — Deux transferts simultanés

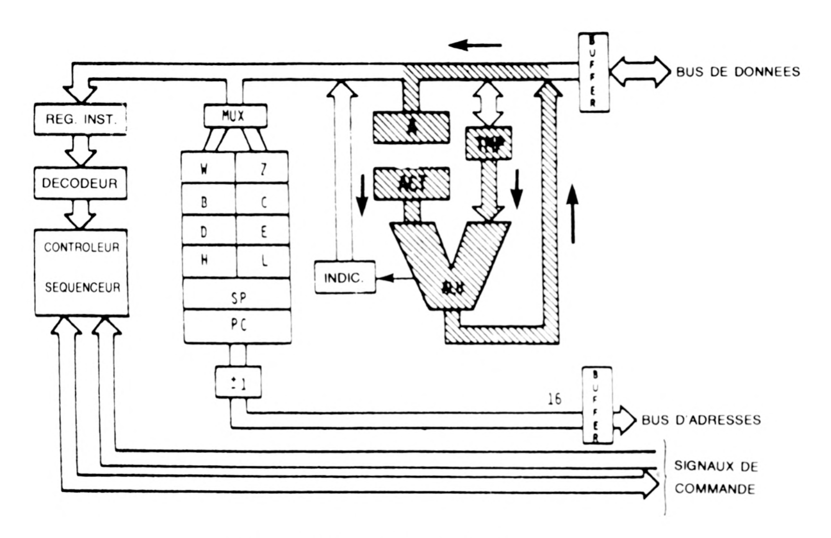


Figure 2.24. — Fin de SUB r

Deux transferts ont lieu simultanément. Le contenu du registre-source spécifié (ici B) est transféré dans TMP, et celui de l'accumulateur A dans l'accumulateur temporaire ACT. En regardant attentivement la figure 2.23., vous vous persuadez que ces deux transferts peuvent avoir lieu simultanément, car ils utilisent des chemins différents. Le transfert de B vers TMP utilise le bus de données interne. Celui de A vers ACT suit un court chemin de données, indépendant du précédent. Ce cheminement parallèle se justifie par la recherche d'un gain de temps. A ce point, les deux entrées de l'ALU sont correctement positionnées. Tout est prêt pour effectuer la soustraction, qui devrait normalement avoir lieu pendant le temps T5 de M1. Or, il n'en est rien. La soustraction ne s'effectue pas ! Nous entrons dans le cycle machine M2. T1 : toujours rien ! C'est seulement au temps T2 du cycle M2 que la soustraction est opérée (cf. Fig. 2.27., instruction SUB r).

T2 du cycle M2 = (ACT) − (TMP) → A

Le contenu de TMP est soustrait de celui de l'accumulateur, et le résultat est finalement déposé dans l'accumulateur. L'opération est terminée (cf. 2.24.)

Question : *Pourquoi la fin de la soustraction a-t'elle été différée ? Pourquoi a-t'elle eu lieu au temps T2 du cycle M2, plutôt qu'au temps T5 de M1 ? (C'est une question difficile, qui suppose une bonne connaissance de l'architecture de la CPU. Cependant, la technique employée est fondamentale, lors de la conception d'une CPU synchronisée sur une horloge. Essayez de comprendre ce qui se passe).*

Réponse : Employée dans de nombreuses CPU, il s'agit d'une « astuce », nommée « recouvrement des temps de récupération et d'exécution », dont l'idée de base est la suivante : l'exécution de la soustraction n'utilise que l'ALU et le bus de données (voir Fig. 2.24). En particulier, cette phase de la soustraction n'utilise ni la RAM interne (le bloc de registres) ni le bus d'adresses. Nous savons (et l'unité de commande aussi) que les trois premiers temps de l'instruction à venir, quelle qu'elle soit, seront les temps T1, T2 et T3 d'un cycle machine M1. Si l'on se remémore la façon dont sont exécutés ces trois temps, on constate que le seul impératif est de disposer du compteur ordinal (PC) et du bus d'adresses. L'accès au compteur ordinal implique l'immobilisation de la RAM. (Ceci explique que cette astuce ne puisse être utilisée pour l'instruction LD R,R'). Il est donc possible d'utiliser simultanément les ressources de la CPU hachurées sur la figure 2.17., et celles hachurées sur la figure 2.24.

Le bus de données est utilisé, pendant le temps T1 de M1, pour transporter des informations concernant l'état du microprocesseur. Il ne peut donc pas l'être pour la soustraction, comme le montre la figure 2.27. Le mécanisme de ce « recouvrement » est maintenant établi. Ses avantages devraient apparaître clairement. Supposons, en effet que nous ayons choisi la voie la plus simple, et effectué la soustraction au cours du temps T5 du cycle machine M1. La durée de la soustraction aurait été 5 × 400 ns = 2 000 ns.

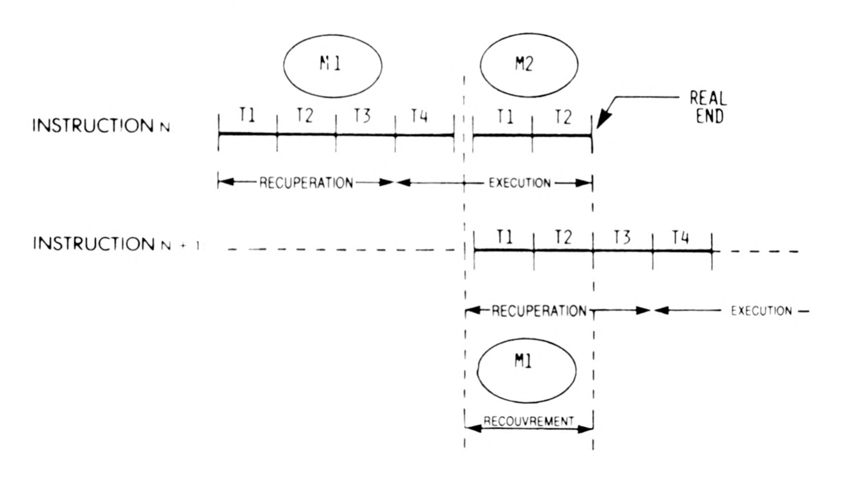


Figure 2.25. — RÉCUPÉRATION-EXÉCUTION : Recouvrement pendant T1 - T2

En utilisant la technique du recouvrement, l'instruction suivante peut démarrer dès que T4 est exécuté. L'astucieuse unité de commande va terminer la soustraction en utilisant le T2 de l'instruction à venir, sans que cette dernière s'en aperçoive. Sur la figure 2.27., T2 fait partie du cycle M2 de la soustraction. Formellement, en effet, il en fait partie. Mais dans la mesure où M2 va se dérouler en même temps que le cycle M1 de

NOTES

1. Le premier cycle mémoire (M1) est toujours une recherche d'instructions ; le premier (ou seul) octet contenant le code opération est récupéré durant ce cycle.

2. Si l'entrée READY en provenance de la mémoire n'est pas haute durant T2 de chaque cycle mémoire, le processeur introduit un temps d'attente (TW) jusqu'à ce que READY remonte.

3. Les temps T4 et T5 sont présents, si nécessaire, pour des opérations internes à la CPU. Durant T4 et T5, le contenu du bus interne est disponible sur le bus de données (ceci uniquement en vue de tests). Un « X » indique que ce temps est présent, mais il est utilisé uniquement pour des opérations internes telles que le décodage d'une instruction.

4. Seules les paires de registres rp = B (registres B et C) ou rp = D (registres D et E) peuvent être spécifiées.

5. Ces temps sont sautés.

6. Sous-cycles de lecture mémoire ; un mot d'instruction ou de données est lu.

7. Sous-cycle d'écriture mémoire.

8. Le signal READY n'est pas nécessaire durant les second et troisième sous-cycles (M2 et M3). Le signal HOLD est accepté durant M2 et M3. Le signal SYNC n'est pas généré durant M2 et M3. Pendant l'exécution de DAD, M2 et M3 sont nécessaires pour une addition interne de paires de registres ; la mémoire n'est pas référencée.

9. Les résultats de ces instructions arithmétiques, logiques ou de rotation ne seront déposés dans l'accumulateur (4) qu'au temps T2 de l'instruction suivante. A est chargé pendant la récupération de l'instruction suivante ; ce recouvrement permet un gain de vitesse.

10. Si la valeur des 4 bits de poids faibles de l'accumulateur est supérieure à 9 ou si l'indicateur auxiliaire de report est à 1, la valeur 6 est ajoutée à l'accumulateur. Si la valeur des 4 bits de poids forts de l'accumulateur devient alors supérieure à 9 ou si l'indicateur de report est à 1, la valeur 6 est ajoutée aux 4 bits de poids forts de l'accumulateur.

11. Premier sous-cycle (récupération de l'instruction) du cycle de l'instruction suivante.

12. Si la condition est vérifiée, c'est le contenu de la paire de registres WZ qui est déposé sur les lignes d'adresses (A_{0-15}) à la place de celui de PC.

13. Si la condition n'est pas vérifiée, les sous-cycles M4 et M5 sont sautés ; à la place, le processeur passe directement à la récupération de l'instruction suivante.

14. Si la condition n'est pas vérifiée, les sous-cycles M2 et M3 sont sautés ; à la place le processeur passe directement à la récupération de l'instruction suivante.

15. Sous-cycle de lecture de pile.

16. Sous-cycle d'écriture de pile.

17. Condition

Condition	CCC
NZ — non zéro (Z = 0)	000
Z — zéro (Z = 1)	001
NC — pas de report (CY = 0)	010
C — report (CY = 1)	011
PO — parité impaire (P = 0)	100
PE — parité paire (P = 1)	101
P — positif (S = 0)	110
M — négatif (S = 1)	111

18. Sous-cycle d'entrée/sortie ; les 8 bits de l'adresse du port d'E/S sont répétés sur les lignes d'adresses 0-7 (A_{0-7}) et 8-15 (A_{8-15}).

19. Sous-cycle d'écriture périphérique.

20. Le processeur restera inactif dans l'état « halt » jusqu'à la réception d'une interruption, d'un reset ou d'un hold. Lorsqu'une requête hold est reçue, le processeur entre dans le mode hold et à la fin retourne à l'état halt. Après un reset, le processeur commence à exécuter des instructions à partir de l'adresse mémoire 0. Après la réception d'une interruption, le processeur exécute l'instruction forcée sur le bus (généralement un restart).

SSS ou DDD	Valeur	rp	Valeur
A	111	B	00
B	000	D	01
C	001	H	10
D	010	SP	11
E	011		
H	100		
L	101		

Figure 2.26. — Abréviations INTEL © Intel-Reproduced by permission

l'instruction à venir, pour le programmeur, la durée de l'instruction SUB r ne dépassera pas : 4 × 400 ns = 1600 ns, soit un gain de 20 % en vitesse. Loin d'être négligeable !

La technique du recouvrement est illustrée à la figure 2.25. On y a recours aussi souvent que possible pour augmenter la vitesse d'exécution apparente du microprocesseur. Mais naturellement, tel n'est pas toujours le cas, car il faut que l'instruction qui se termine et celle qui commence n'aient pas besoin simultanément des mêmes ressources de la CPU (bus par exemple).

Question : *Est-il possible d'aller plus loin dans l'exploitation de la technique de recouvrement, et d'utiliser aussi le temps T3 de M2 dans le cas d'une instruction plus longue ?*

Pour une vision plus claire des mécanismes internes de séquencement, il est recommandé d'examiner attentivement la figure 2.27, qui montre le

détail de l'exécution des instructions du 8080. Le Z80 comprend parmi beaucoup d'autres toutes les instructions du 8080. Mais les deux microordinateurs n'exécutent pas les instructions exactement de la même manière. La figure 2.27. n'est, par exemple, pas valable pour le Z80, et n'est exposée que dans un but pédagogique. La correspondance entre les mnémoniques Z80 et 8080, pour les instructions qui leur sont communes, est présentée dans les appendices F et G.

MNEMONIC	OP CODE		M1[1]					M2		
	$D_7 D_6 D_5 D_4$	$D_3 D_2 D_1 D_0$	T1	T2[2]	T3	T4	T5	T1	T2[2]	T3
MOV r1, r2	0 1 D D	D S S S	PC OUT STATUS	PC = PC +1	INST→TMP/IR	(SSS)→TMP	(TMP)→DDD			
MOV r, M	0 1 D D	D 1 1 0				X[3]		HL OUT STATUS[6]	DATA →	DDD
MOV M, r	0 1 1 1	0 S S S				(SSS)→TMP		HL OUT STATUS[7]	(TMP) →	DATA BUS
SPHL	1 1 1 1	1 0 0 1				(HL) →	SP			
MVI r, data	0 0 D D	D 1 1 0				X		PC OUT STATUS[6]	B2 →	DDDD
MVI M, data	0 0 1 1	0 1 1 0				X			B2 →	TMP
LXI rp, data	0 0 R P	0 0 0 1				X			PC = PC + 1 B2 →	r1
LDA addr	0 0 1 1	1 0 1 0				X			PC = PC + 1 B2 →	Z
STA addr	0 0 1 1	0 0 1 0				X			PC = PC + 1 B2 →	Z
LHLD addr	0 0 1 0	1 0 1 0				X			PC = PC + 1 B2 →	Z
SHLD addr	0 0 1 0	0 0 1 0				X		PC OUT STATUS[6]	PC = PC + 1 B2 →	Z
LDAX rp[4]	υ 0 R P	1 0 1 0				X		rp OUT STATUS[6]	DATA →	A
STAX rp[4]	0 0 R P	0 0 1 0				X		rp OUT STATUS[7]	(A) →	DATA BUS
XCHG	1 1 1 0	1 0 1 1				(HL)↔(DE)				
ADD r	1 0 0 0	0 S S S				(SSS)→TMP (A)→ACT		[9]	(ACT)+(TMP)→A	
ADD M	1 0 0 0	0 1 1 0				(A)→ACT		HL OUT STATUS[6]	DATA →	TMP
ADI data	1 1 0 0	0 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2 →	TMP
ADC r	1 0 0 0	1 S S S				(SSS)→TMP (A)→ACT		[9]	(ACT)+(TMP)+CY→A	
ADC M	1 0 0 0	1 1 1 0				(A)→ACT		HL OUT STATUS[6]	DATA →	TMP
ACI data	1 1 0 0	1 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2 →	TMP
SUB r	1 0 0 1	0 S S S				(SSS)→TMP (A)→ACT		[9]	(ACT)-(TMP)→A	
SUB M	1 0 0 1	0 1 1 0				(A)→ACT		HL OUT STATUS[6]	DATA →	TMP
SUI data	1 1 0 1	0 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2 →	TMP
SBB r	1 0 0 1	1 S S S				(SSS)→TMP (A)→ACT		[9]	(ACT)-(TMP)-CY→A	
SBB M	1 0 0 1	1 1 1 0				(A)→ACT		HL OUT STATUS[6]	DATA →	TMP
SBI data	1 1 0 1	1 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2 →	TMP
INR r	0 0 D D	D 1 0 0				(DDD)→TMP (TMP) + 1→ALU	ALU→DDD			
INR M	0 0 1 1	0 1 0 0				X		HL OUT STATUS[6]	DATA → (TMP)+1 →	TMP ALU
DCR r	0 0 D D	D 1 0 1				(DDD)→TMP (TMP)+1→ALU	ALU→DDD			
DCR M	0 0 1 1	0 1 0 1				X		HL OUT STATUS[6]	DATA → (TMP)-1 →	TMP ALU
INX rp	0 0 R P	0 0 1 1				(RP) + 1 →	RP			
DCX rp	0 0 R P	1 0 1 1				(RP) - 1 →	RP			
DAD rp[8]	0 0 R P	1 0 0 1				X		(ri)→ACT	(L)→TMP, (ACT)+(TMP)→ALU	ALU→L, CY
DAA	0 0 1 0	0 1 1 1				DAA→A, FLAGS[10]				
ANA r	1 0 1 0	0 S S S				(SSS)→TMP (A)→ACT		[9]	(ACT)+(TMP)→A	
ANA M	1 0 1 0	0 1 1 0	PC OUT STATUS	PC = PC + 1	INST→TMP/IR	(A)→ACT		HL OUT STATUS[6]	DATA →	TMP

Figure 2.27. — Format des instructions INTEL

M3			M4			M5				
T1	T2[2]	T3	T1	T2[2]	T3	T1	T2[2]	T3	T4	T5
HL OUT STATUS[7]	(TMP) →	DATA BUS								
PC OUT STATUS[6]	PC = PC + 1 B3 →	rh								
↑	PC = PC + 1 B3 →	W	WZ OUT STATUS[6]	DATA →	A					
│	PC = PC + 1 B3 →	W	WZ OUT STATUS[7]	(A) →	DATA BUS					
↓	PC = PC + 1 B3 →	W	WZ OUT STATUS[6]	DATA → WZ = WZ + 1	L	WZ OUT STATUS[6]	DATA →	H		
PC OUT STATUS[6]	PC = PC + 1 B3 →	W	WZ OUT STATUS[7]	(L) → WZ = WZ + 1	DATA BUS	WZ OUT STATUS[7]	(H) →	DATA BUS		
[9]	(ACT)+(TMP)→A									
[9]	(ACT)+(TMP)→A									
[9]	(ACT)+(TMP)+CY→A									
[9]	(ACT)+(TMP)+CY→A									
[9]	(ACT)-(TMP)→A									
[9]	(ACT)-(TMP)→A									
[9]	(ACT)-(TMP)-CY→A									
[9]	(ACT)-(TMP)-CY→A									
HL OUT STATUS[7]	ALU →	DATA BUS								
HL OUT STATUS[7]	ALU →	DATA BUS								
(rh)→ACT	(H)→TMP (ACT)+(TMP)+CY→ALU	ALU→H, CY								
[9]	(ACT)+(TMP)→A									

Figure 2.27. — Format des instructions INTEL (suite)

MNEMONIC	OP CODE		M1[1]					M2		
	$D_7 D_6 D_5 D_4$	$D_3 D_2 D_1 D_0$	T1	T2[2]	T3	T4	T5	T1	T2[2]	T3
ANI data	1 1 1 0	0 1 1 0	PC OUT STATUS	PC = PC + 1	INST→TMP/IR	(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2	→ TMP
XRA r	1 0 1 0	1 S S S				(A)→ACT (SSS)→TMP		[9]	(ACT)+(TPM)→A	
XRA M	1 0 1 0	1 1 1 0				(A)→ACT		HL OUT STATUS[6]	DATA	→ TMP
XRI data	1 1 1 0	1 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2	→ TMP
ORA r	1 0 1 1	0 S S S				(A)→ACT (SSS)→TMP		[9]	(ACT)+(TMP)→A	
ORA M	1 0 1 1	0 1 1 0				(A)→ACT		HL OUT STATUS[6]	DATA	→ TMP
ORI data	1 1 1 1	0 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2	→ TMP
CMP r	1 0 1 1	1 S S S				(A)→ACT (SSS)→TMP		[9]	(ACT)-(TMP), FLAGS	
CMP M	1 0 1 1	1 1 1 0				(A)→ACT		HL OUT STATUS[6]	DATA	→ TMP
CPI data	1 1 1 1	1 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2	→ TMP
RLC	0 0 0 0	0 1 1 1				(A)→ALU ROTATE		[9]	ALU→A, CY	
RRC	0 0 0 0	1 1 1 1				(A)→ALU ROTATE		[9]	ALU→A, CY	
RAL	0 0 0 1	0 1 1 1				(A), CY→ALU ROTATE		[9]	ALU→A, CY	
RAR	0 0 0 1	1 1 1 1				(A), CY→ALU ROTATE		[9]	ALU→A, CY	
CMA	0 0 1 0	1 1 1 1				$(\bar{A})$→A				
CMC	0 0 1 1	1 1 1 1				$\overline{CY}$→CY				
STC	0 0 1 1	0 1 1 1				1→CY				
JMP addr	1 1 0 0	0 0 1 1				X		PC OUT STATUS[6]	PC = PC + 1 B2	→ Z
J cond addr[17]	1 1 C C	C 0 1 0				JUDGE CONDITION		PC OUT STATUS[6]	PC = PC + 1 B2	→ Z
CALL addr	1 1 0 0	1 1 0 1				SP = SP - 1		PC OUT STATUS[6]	PC = PC + 1 B2	→ Z
C cond addr[17]	1 1 C C	C 1 0 0				JUDGE CONDITION IF TRUE, SP = SP - 1		PC OUT STATUS[6]	PC = PC + 1 B2	→ Z
RET	1 1 0 0	1 0 0 1				X		SP OUT STATUS[15]	SP = SP + 1 DATA	→ Z
R cond addr[17]	1 1 C C	C 0 0 0			INST→TMP/IR	JUDGE CONDITION[14]		SP OUT STATUS[15]	SP = SP + 1 DATA	→ Z
RST n	1 1 N N	N 1 1 1			φ→W INST→TMP/IR	SP = SP - 1		SP OUT STATUS[16]	SP = SP - 1 (PCH)	→ DATA BUS
PCHL	1 1 1 0	1 0 0 1			INST→TMP/IR	(HL)	→ PC			
PUSH rp	1 1 R P	0 1 0 1				SP = SP - 1		SP OUT STATUS[16]	SP = SP - 1 (rh)	→ DATA BUS
PUSH PSW	1 1 1 1	0 1 0 1				SP = SP - 1		SP OUT STATUS[16]	SP = SP - 1 (A)	→ DATA BUS
POP rp	1 1 R P	0 0 0 1				X		SP OUT STATUS[15]	SP = SP + 1 DATA	→ r1
POP PSW	1 1 1 1	0 0 0 1				X		SP OUT STATUS[15]	SP = SP + 1 DATA	→ FLAGS
XTHL	1 1 1 0	0 0 1 1				X		SP OUT STATUS[15]	SP = SP + 1 DATA	→ Z
IN port	1 1 0 1	1 0 1 1				X		PC OUT STATUS[6]	PC = PC + 1 B2	→ Z, W
OUT port	1 1 0 1	0 0 1 1				X		PC OUT STATUS[6]	PC = PC + 1 B2	→ Z, W
EI	1 1 1 1	1 0 1 1				SET INTE F/F				
DI	1 1 1 1	0 0 1 1				RESET INTE F/F				
HLT	0 1 1 1	0 1 1 0				X		PC OUT STATUS	HALT MODE[20]	
NOP	0 0 0 0	0 0 0 0	PC OUT STATUS	PC = PC + 1	INST→TMP/IR	X				

Figure 2.27[1]. — Format des instructions INTEL (suite)

M3			M4			M5						
T1	T2[2]	T3	T1	T2[2]	T3	T1	T2[2]	T3	T4	T5		
[9]	(ACT)+(TMP)→A											
[9]	(ACT)+(TMP)→A											
[9]	(ACT)+(TMP)→A											
[9]	(ACT)+(TMP)→A											
[9]	(ACT)+(TMP)→A											
[9]	(ACT)-(TMP); FLAGS											
[9]	(ACT)-(TMP); FLAGS											
PC OUT STATUS[6]	PC = PC + 1 B3 →	W									WZ OUT STATUS[11]	(WZ) + 1 → PC
PC OUT STATUS[6]	PC = PC + 1 B3 →	W									WZ OUT STATUS[11,12]	(WZ) + 1 → PC
PC OUT STATUS[6]	PC = PC + 1 B3 →	W	SP OUT STATUS[16]	(PCH) → SP = SP - 1	DATA BUS	SP OUT STATUS[16]	(PCL) →	DATA BUS			WZ OUT STATUS[11]	(WZ) + 1 → PC
PC OUT STATUS[6]	PC = PC + 1 B3 →	W[13]	SP OUT STATUS[16]	(PCH) → SP = SP - 1	DATA BUS	SP OUT STATUS[16]	(PCL) →	DATA BUS			WZ OUT STATUS[11,12]	(WZ) + 1 → PC
SP OUT STATUS[15]	SP = SP + 1 DATA →	W									WZ OUT STATUS[11]	(WZ) + 1 → PC
SP OUT STATUS[15]	SP = SP + 1 DATA →	W									WZ OUT STATUS[11,12]	(WZ) + 1 → PC
SP OUT STATUS[16]	(TMP = 00NNN000) → (PCL) →	Z DATA BUS									WZ OUT STATUS[11]	(WZ) + 1 → PC
SP OUT STATUS[16]	(rl) →	DATA BUS										
SP OUT STATUS[16]	FLAGS →	DATA BUS										
SP OUT STATUS[15]	SP = SP + 1 DATA →	rh										
SP OUT STATUS[15]	SP = SP + 1 DATA →	A										
SP OUT STATUS[15]	DATA →	W	SP OUT STATUS[16]	(H) →	DATA BUS	SP OUT STATUS[16]	(L) →	DATA BUS	(WZ) →	HL		
WZ OUT STATUS[18]	DATA →	A										
WZ OUT STATUS[18]	(A) →	DATA BUS										

Figure 2.27[1]. — Format des instructions INTEL

Examinons maintenant une instruction plus complexe :

ADD A, (HL)

Le code-opération de cette instruction est 10000110. Sa signification est : « additionner au contenu de l'accumulateur celui de l'emplacement-mémoire dont l'adresse est donnée par HL, puis mettre le résultat dans l'accumulateur ». L'adresse-mémoire concernée est ici spécifiée de manière bien étrange. Cette instruction suppose que deux registres, H et L, ont été chargés avant son exécution. Le contenu, sur 16 bits, de ces deux registres regroupés en paire (HL), est une adresse-mémoire. Le contenu de l'emplacement-mémoire possédant cette adresse sera ajouté à celui de l'accumulateur, et le résultat sera laissé dans l'accumulateur.

Cette instruction a une histoire. Son rôle fut jadis d'établir une compatibilité entre le vieux 8008 et son successeur, le 8080. Le 8008 ne connaissait pas l'adressage direct de la mémoire ! Pour accéder au contenu d'un emplacement-mémoire, il fallait mettre son adresse dans les deux registres H et L, puis exécuter une instruction utilisant H et L.

ADD A, (HL) est un bon exemple de ce type d'instructions. Il faut bien préciser que, ni le 8080, ni le Z 80, ne sont soumis aux mêmes limitations, concernant l'adressage de la mémoire. Tous deux permettent un adressage direct. Dans ces conditions, la possibilité d'utiliser aussi l'adressage via H et L devient un avantage supplémentaire, au lieu de n'être qu'un pis-aller, comme pour le 8008.

Suivons maintenant l'exécution de cette instruction (dans le 8080, elle s'appelle ADD M. C'est la 16e instruction de la figure 2.27). On se sert des temps T1, T2 et T3 de manière habituelle, pour récupérer l'instruction. Pendant le temps T4, le contenu de l'accumulateur est transféré dans son registre tampon ACT, fournissant ainsi l'entrée gauche de l'ALU.

On aura maintenant recours à un accès-mémoire pour récupérer l'octet à ajouter à l'accumulateur. L'adresse de cet octet est contenue dans les registres H et L. Le contenu de ces registres sera placé sur le bus d'adresses, avant d'être émis vers la mémoire.

Sur la figure 2.27, au cycle-machine M2, nous lisons : HL OUT (sortir HL). H et L sont déposés sur le bus d'adresses, de la même façon que PC lors des précédentes instructions. Remarque : nous avons déjà souligné que le temps T1 utilise le bus de données pour y déposer des informations concernant l'état du microprocesseur. Nous n'utiliserons pas ici ces informations. Pour simplifier, disons que deux temps seront nécessaires : un pour permettre à la mémoire de trouver l'information demandée, et un autre pour lui permettre de la fournir, de telle sorte qu'elle puisse être déposée à l'entrée droite de l'ALU, dans le registre TMP.

Les deux entrées de l'ALU sont maintenant positionnées. Nous nous trouvons dans une situation absolument analogue à celle de la précédente instruction, SUB r (à ceci près qu'il s'agit d'une addition, mais nous avons vu, au chapitre I, que la représentation en complément à 2 n'implique pas de distinction). Il ne reste donc plus qu'à effectuer l'addition, comme

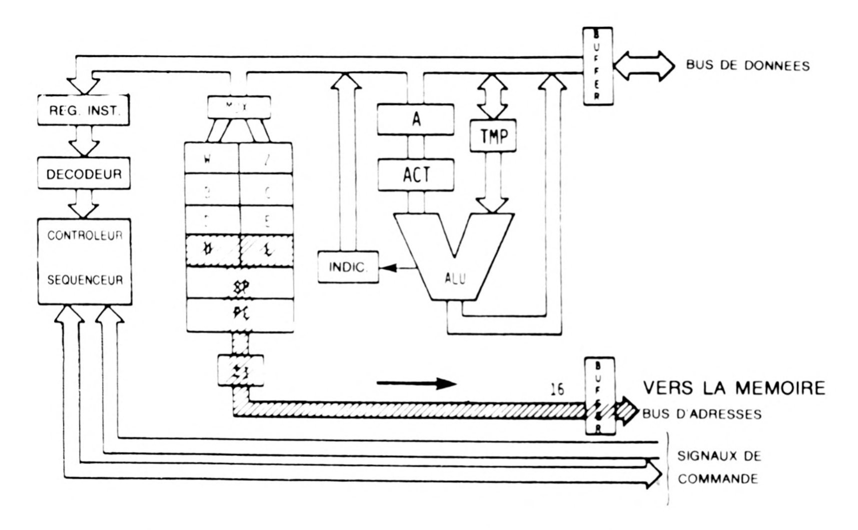


Figure 2.28. — Transférer le contenu de HL sur le bus d'adresses

précédemment la soustraction. La technique du recouvrement sera à nouveau utilisée, et la fin de l'exécution sera reportée au temps de T2 de M3, au lieu de prendre place au temps T4 de M2. La figure 2.27 montre qu'au temps T2 de M3 : ACT + TMP → A. L'addition est enfin effectuée, le contenu de ACT est ajouté à celui de TMP et le résultat est déposé dans l'accumulateur A.

Question : *Quel est pour le programmeur le temps apparent d'exécution de cette instruction ? Est-ce 3,6 μs ou 2,8 μs. (En prenant toujours une fréquence d'horloge de 2,5 MHz) ?*

Une instruction encore un peu plus compliquée va maintenant être présentée. Elle réalise un adressage direct de la mémoire, et utilise deux registres inaccessibles au programmeur : W et Z.

LD A, (nn)

Son code-opération est 00111010. L'équivalent 8080 est LDA addr. Comme d'habitude, les temps T1, T2 et T3 sont utilisés à la récupération, depuis la mémoire, du code-opération de l'instruction. T4 est également utilisé. Rien de visible n'y a lieu, mais en réalité, pendant ce laps de temps, l'instruction est décodée. L'unité de contrôle s'aperçoit alors qu'elle doit aller chercher les deux octets suivants de l'instruction. Ces octets préciseront l'adresse mémoire, dont il restera à obtenir le contenu. Remarquons que le temps T4 est nécessaire au décodage de l'instruction. C'est étonnant, dans la mesure où nous étions habitués à ce que le décodage ait lieu uniquement pendant T3. Cette anomalie tient, en fait, à la philosophie qui a

présidé à la conception de la plupart des microprocesseurs : synchronisation des opérations sur une horloge, un événement par trop d'horloge. Autrement dit faire en sorte que toutes les instructions soient décodées en un seul temps de l'horloge. A l'intérieur du microprocesseur, des *microinstructions* sont utilisées pour ce décodage. Certains codes-opérations demandent l'exécution d'un plus grand nombre de microinstructions. Les avantages de la microprogrammation se paient, parfois. En fait, lors de la conception d'un microprocesseur, des efforts importants sont faits pour que les instructions exigeant un long décodage soient, ou bien des instructions rarement utilisées, ou bien surtout, et c'est le cas pour LD A, (nn), des instructions nécessitant un deuxième cycle machine, juste après le décodage. T4 de M1 n'étant pas, dans ce cas, mis à profit.

Le format de l'instruction apparaît à la figure 2.29.

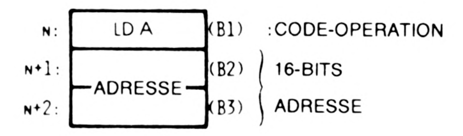


Figure 2.29. — LD A, (ADRESSE) est une instruction de 3 mots

Les deux octets suivants de l'instruction vont maintenant être recherchés (voir figure 2.30).

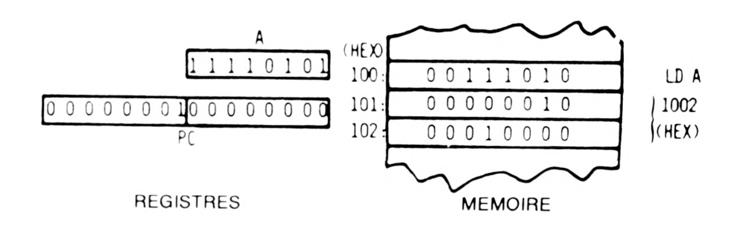


Figure 2.30. — Avant l'exécution de LD A, (nn)

L'effet de cette instruction est présenté sur les figures 2.30 et 2.31.

L'unité de commande du Z80 a accès à deux registres spéciaux, W et Z, que le programmeur ne peut utiliser (cf. figure 2.28).

Cycle machine M2 : Comme toujours les temps T1 et T2 sont mis à profit pour rechercher le contenu de l'adresse mémoire spécifiée par PC. Pendant

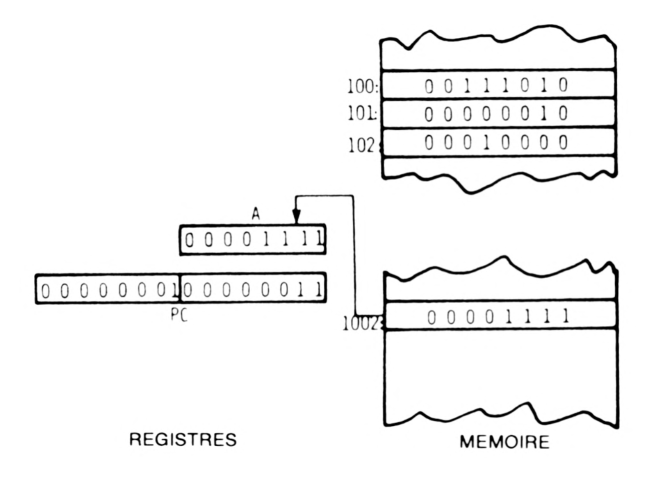


Figure 2.31. — Après l'exécution de LD A, (nn)

T2, le contenu de PC est incrémenté de 1. Vers la fin de cette période, l'octet en provenance de la mémoire apparaît sur le bus de données. Au milieu de T3, le Z80 le lit (= le transfère sur son bus interne). Il convient maintenant de ranger ce second octet de l'instruction (B2) dans un registre temporaire. Il est déposé dans Z : B2 → Z (voir figure 2.32).

Cycle machine M3 : de nouveau, le contenu de PC est déposé sur le bus d'adresses, incrémenté pour la fois suivante, et finalement, le troisième octet (B3) de l'instruction est lu et déposé dans le registre W du microprocesseur. A ce point, les registres W et Z du microprocesseur contiennent B2 et B3, c'est-à-dire l'adresse sur 16 bits présente, au départ,

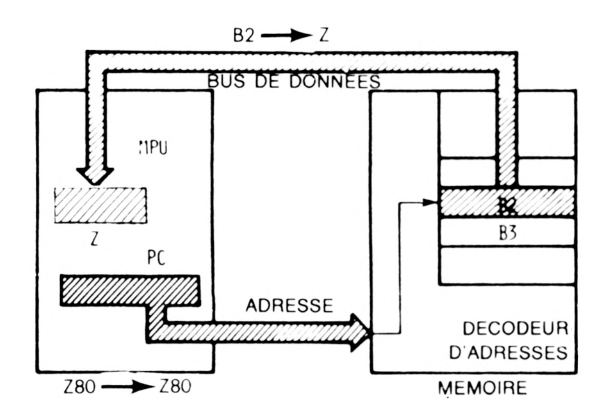


Figure 2.32. — Le second octet de l'instruction est rangé dans Z

en mémoire, juste derrière le code opération de l'instruction. Nous pouvons maintenant terminer l'exécution. W et Z contiennent une adresse qui doit être envoyée à la mémoire pour que cette dernière en fournisse, en retour, le contenu, lors du cycle machine suivant.

Cycle machine M4 : Cette fois, les contenus de W et Z sont déposés sur le bus d'adresses. Vers la fin de T2, le contenu de l'adresse impliquée devient disponible. Il est finalement lu et déposé dans l'accumulateur, vers la fin de T3. Ainsi, se termine l'exécution de cette instruction.

Il s'agit là d'un bon exemple d'*instruction explicite.* Elle nécessite 3 octets, dont ceux pour y ranger une *adresse explicite,* et utilise quatre cycles-mémoire : trois pour obtenir les 3 octets de l'instruction, plus un pour récupérer l'octet spécifié par l'adresse nn. C'est une instruction longue, mais cependant essentielle, puisqu'elle permet de charger l'accumulateur avec le contenu d'un emplacement mémoire connu. Notons aussi qu'elle utilise les registres W et Z.

Question : *Cette instruction pouvait-elle utiliser d'autres registres du microprocesseur que W et Z ?*

Réponse : Non. Elle n'aurait pu utiliser d'autres registres, par exemple H et L, sans en modifier le contenu. A la fin de l'exécution, les anciens contenus de H et de L auraient été perdus. Dans un programme, on avance toujours l'hypothèse qu'une instruction ne modifie pas d'autre registre que ceux qu'elle est censée utiliser. Une instruction qui charge l'accumulateur ne doit pas détruire le contenu d'un autre registre. Pour cette raison, il était nécessaire de disposer des deux registres supplémentaires, W et Z.

Question : *Etait-il possible d'utiliser PC, au lieu de W et Z ?*

Réponse : Absolument pas. Ce serait du suicide. Le lecteur est invité à justifier cette réponse.

Voyons maintenant un autre type : l'instruction de *branchement* ou de *saut,* qui permet de modifier l'ordre d'exécution des instructions. Ordre supposé, jusqu'ici, séquentiel, les instructions étant exécutées les unes après les autres. Il existe pourtant des instructions permettant au programmeur de rompre cette séquence, soit pour exécuter une autre séquence d'instructions, soit plus concrètement, pour aller vers une autre zone de la mémoire contenant le programme. L'exemple d'une telle instruction est :

JP nn

Elle est l'équivalent de l'instruction 8080 « JMP addr », à la ligne 18 de la deuxième partie de la figure 2.27[1]. Son exécution va être expliquée en suivant pas à pas la ligne horizontale de la figure. Il s'agit d'une instruction de trois mots. Le premier est le code opération : il vaut 11000011. Les deux suivants contiennent, sur 16 bits, l'adresse où aura lieu le saut. Vu de l'extérieur, l'effet de cette instruction est de remplacer le contenu du

compteur ordinal par les 16 bits succédant au code opération de l'instruction JP (Jump : saut). En pratique, l'instruction se déroule, pour des raisons d'efficacité, de manière légèrement différente.

Comme d'habitude, les trois premiers temps de M1 correspondent à la récupération de l'instruction. Pendant T4, l'instruction est décodée. On n'enregistre aucun autre événement (ce qui est figuré par un X sur la figure 2.27). Les deux cycles-machine suivants servent à la récupération des octets B2 et B3 de l'instruction. Pendant M2, B2 est récupéré, et déposé dans le registre interne Z. Pendant M3, B3 est récupéré, et déposé dans W. Les deux étapes ultérieures se déroulent pendant la phase de récupération de l'instruction suivante. Même chose, donc, que pour la soustraction. A ceci près que dans une instruction de saut, elles ne coïncident pas avec les temps T1 et T2 du cycle M1 de l'instruction suivante, mais ont lieu *à leur place*. Voyons de plus près ce qui se passe.

Après le cycle M3 de l'instruction JP, le registre WZ contient l'adresse où l'instruction à venir devra être recherchée. Plutôt que d'ajouter un cycle-machine supplémentaire, destiné à transférer le contenu de WZ dans PC, le contenu de WZ sera directement déposé sur le bus d'adresses, puis incrémenté de 1, avant d'être déposé dans le compteur ordinal. (Voir la partie droite Jump : WZ OUT et (WZ) + 1 → PC).

L'unité de commande sait qu'elle vient d'exécuter l'instruction Jump, et modifie en conséquence le cycle M1 de l'instruction suivante. Dans la mesure où le contenu, incrémenté de 1, de WZ est déposé dans le compteur ordinal, l'unité de commande pourra poursuivre la recherche des instructions à l'endroit de la mémoire spécifié dans l'instruction Jump.

Bien que l'implémentation de cette instruction soit un peu différente de son rôle apparent, le résultat recherché (le saut) est malgré tout obtenu.

Question : *Pourquoi le compteur ordinal n'a-t-il pas été chargé directement ? Pourquoi avoir utilisé les registres intermédiaires W et Z ?*

Réponse : Il n'était pas possible d'utiliser PC. Si le deuxième octet de l'instruction (B2) avait été chargé dans la partie basse du compteur ordinal, au lieu de l'être dans Z, PC aurait été détruit. Plus question, alors, d'aller rechercher B3.

Question : *Est-il possible d'utiliser le seul registre Z, au lieu de W et Z ?*

Réponse : Oui, mais l'opération est plus lente. Il était, en effet, possible d'aller chercher B2, de le déposer dans Z, et de renouveler la manœuvre avec B3, déposé dans la partie haute du compteur ordinal. Cependant, il aurait alors fallu transférer Z dans la partie basse de PC, avant de pouvoir utiliser ce dernier. Le processus aurait été ralenti. Voilà pourquoi il est préférable d'utiliser W et Z. Bien plus, c'est encore pour gagner du temps que le contenu de WZ n'est pas transféré dans PC, mais directement déposé sur le bus d'adresses, de façon à aller rechercher l'instruction suivante. Il est important de bien comprendre ce point pour apprécier l'efficacité de l'exécution des instructions dans le microprocesseur.

Question : *(Pour le lecteur déjà informé seulement).*
Que se passerait-il si une interruption avait lieu à la fin du cycle M3 ? (Si l'exécution est suspendue à cet endroit, le compteur ordinal pointe sur l'instruction qui suit le saut. L'adresse à laquelle le programme doit se débrancher est perdue).

La réponse à cet intéressant exercice est laissée à la perspicacité du lecteur.

La description détaillée que nous avons présentée de quelques instructions-type, devrait avoir précisé le rôle et le fonctionnement des registres et des bus internes. Une seconde lecture de ce chapitre est susceptible d'améliorer encore la connaissance que l'on vient d'acquérir du fonctionnement interne d'un microprocesseur tel que le Z80. Il convient de repréciser ici que le répertoire des instructions fournies est celui du 8080. Les principes de fonctionnement du Z80 sont les mêmes, mais les améliorations qu'il introduit ont conduit ses concepteurs à modifier légèrement l'exécution de certaines instructions. Le lecteur avisé aura déjà remarqué que l'instruction LD r,r' du Z80 n'utilise que quatre temps d'horloge apparents, contre cinq pour le 8080, tandis que l'instruction « INX rp » du 8080 (augmenter de 1 le contenu d'une paire de registres internes) utilise cinq temps d'horloge apparents, quand son équivalent Z80, « INC ss », en utilise six.

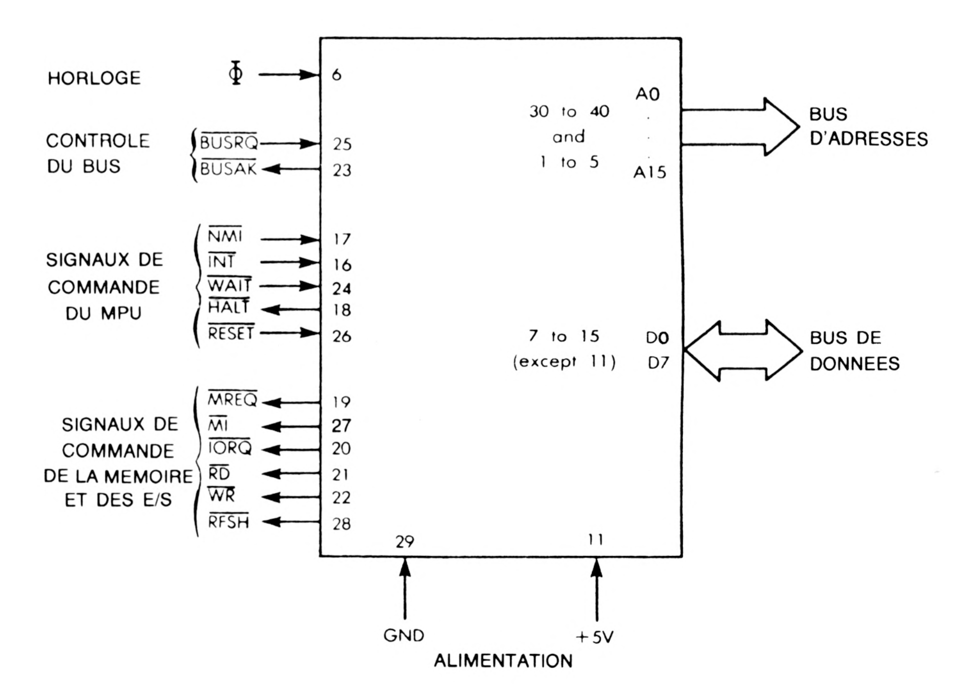


Figure 2.33. — Les broches du MPU Z80

Le boitier Z80

Pour achever notre description, nous allons maintenant examiner les signaux qui entrent dans le Z80, ou qui en sortent. Il n'est pas indispensable de comprendre la fonction des signaux du Z80 pour le programmer. Le lecteur peu intéressé par les détails hardware peut donc, sans dommage, sauter ce paragraphe. Le brochage du Z80 est montré à la figure 2.33. Sur la partie droite du dessin sont représentés le bus d'adresses et le bus de données (dont le comportement a été expliqué). Nous nous attacherons ici au rôle des signaux du bus de commande, représentés sur la partie gauche du dessin.

Les signaux de commande ont été répartis en quatre groupes. Ils seront décrits un par un, en suivant de haut en bas la figure 2.33.

L'entrée de l'horloge s'appelle Φ. Seule nécessité : une résistance externe de 330 Ω, montée en « pull-up » entre l'horloge et le + 5V. Cependant, pour l'usage des boitiers Z80-A qui admettent une horloge de 4 MHz, une mise en forme préalable du signal d'horloge doit être entreprise, au moyen d'un « driver ».

Les deux signaux du groupe « commande des bus », BUSRQ et BUSAK, servent à déconnecter le Z80 de ses bus. Ils sont principalement utilisés par les circuits d'accès direct mémoire (DMA), mais peuvent également l'être par un second microprocesseur monté dans le système.

En réponse à une demande extérieure de déconnexion formulée par l'activation du signal BUSRQ, le Z80, à la fin du cycle-machine en cours, met tous ses signaux de sortie trois-états en haute impédance, puis active le signal BUSAK pour signaler au demandeur qu'il s'est déconnecté de ses bus, et que ce dernier peut donc les utiliser.

Six signaux de commande sont liés à l'état du microprocesseur, ou au séquencement des instructions.

INT et NMI sont deux signaux d'interruption ; le premier étant le signal d'interruption habituel (voir chapitre 6). Un certain nombre de périphériques d'entrée-sortie peuvent être connectés au signal INT. Chaque fois qu'une interruption est demandée sur cette ligne, et que la bascule interne IFF le permet, le Z80 l'accepte (à condition, toutefois, que le signal BUSRQ soit inactif). Il génère alors un signal d'acquiescement en activant simultanément les signaux M1 et IORQ. La suite de ce processus sera expliquée au chapitre 6.

NMI est le signal d'interruption non-masquable. Il est toujours accepté par le Z80, qu'il force à exécuter le programme situé à l'adresse 102 (66 hexadécimal). Là encore, l'acceptation est conditionnée à l'inactivité de BUSRQ. Le fonctionnement de NMI est décrit au chapitre 6.

WAIT est un signal permettant de ralentir le Z80 lorsqu'il échange des informations avec des mémoires ou des périphériques lents.

Activé, ce signal indique que la mémoire ou le périphérique n'est pas prêt à procéder à l'échange. Le Z80 se met alors dans un état spécial d'attente jusqu'à ce que le signal soit désactivé. Il reprend alors son fonctionnement normal.

Le Z80 émet un signal HALT pour indiquer qu'il vient d'exécuter l'instruction HALT. Il ne peut repartir qu'après réception d'une interruption. En attendant, il exécute sans arrêt des instructions NOP (de l'anglais No OPeration, aucune opération).

L'exécution continuelle de ces NOP permet de continuer, quand même, à rafraîchir les mémoires dynamiques. En effet, pendant les temps T3 et T4 du cycle M1 de l'exécution d'une instruction, le Z80 n'utilise jamais le bus d'adresses. Il en profite donc pour envoyer aux mémoires des signaux pour leur indiquer qu'il ne désire pas les utiliser, et leur permettre de mettre à profit ces temps « morts » pour se rafraîchir.

RESET est le signal destiné à initialiser le fonctionnement du MPU. Celui-ci met le compteur ordinal et les registres I et R à zéro, s'interdit d'accepter des interruptions et se met en mode d'interruption « 0 ». Ce signal est normalement utilisé à la mise sous tension du système.

Signaux de commandes de la mémoire et des entrées/sorties

Six signaux de contrôle de la mémoire, de même que des E/S (entrées-sorties), sont générés par le Z80.

MREQ (de l'anglais Memory REQuest) est le signal indiquant que le Z80 va requérir l'usage de la mémoire. Il signale qu'une adresse mémoire valide est présente sur le bus d'adresses. Une opération de lecture, ou d'écriture, pourra alors avoir lieu.

M1 est un signal actif pendant le cycle machine M1 de chaque instruction, au cours duquel le premier octet est récupéré.

IORQ indique que le Z80 veut effectuer une opération d'entrée-sortie. Ce signal précise qu'une adresse d'E/S est présente sur les bits 0 à 7 du bus d'adresses. Une opération d'E/S pourra alors être effectuée. IORQ et M1 peuvent être activés simultanément et indiquent, dans ce cas, que le Z80 a détecté une interruption, et accepte de la prendre en compte. Cette information peut être utilisée par les boitiers périphériques pour déposer leur vecteur d'interruption sur le bus de données. (Aucune opération normale d'E/S ne peut avoir lieu pendant le cycle M1 d'une instruction ; c'est pourquoi la conjonction des deux signaux M1 et IORQ prend une signification particulière).

RD est le signal de lecture. Il indique que le Z80 est prêt à lire le contenu du bus de données, dans l'un de ses registres internes. Il peut être utilisé par n'importe quel boitier du système [boitier mémoire ou d'E/S] pour déposer des données sur le bus de données.

WR est le signal d'écriture. Il porte à la connaissance des autres boitiers du système que l'information déposée par le MPU sur le bus de données est valide, prête à être écrite dans le boitier destinataire.

Enfin, RFSH est le signal de rafraîchissement destiné aux mémoires dynamiques. Lorsqu'il est actif, les sept bits de poids faibles du bus d'adresses contiennent une adresse de rafraîchissement. Le signal MREQ est ensuite envoyé en vue d'une lecture de rafraîchissement.

CONCLUSION SUR LE HARDWARE

Ainsi se termine notre description de l'organisation interne du Z80. Les détails précis du hardware n'importent pas ici, à l'exception de la fonction de chaque registre. Cette dernière doit être parfaitement comprise avant d'aborder les chapitres suivants. Nous allons maintenant présenter le jeu d'instructions et les techniques de base de la programmation du Z80.

3

TECHNIQUES DE BASE DE LA PROGRAMMATION

INTRODUCTION

Notre propos est de présenter les techniques de base nécessaires, sur le Z80, à l'écriture d'un programme. Ce chapitre va introduire de nouveaux concepts, tels que la gestion des registres, les boucles et les sous-programmes. Il s'attachera particulièrement aux techniques utilisant les seules ressources internes du Z80, c'est-à-dire les registres. Nous développerons de véritables programmes, en particulier arithmétiques, qui serviront d'illustration aux concepts divers présentés jusqu'ici, et utiliseront des instructions réelles. Ainsi, nous examinerons la manière dont les instructions peuvent être utilisées pour échanger l'information entre la mémoire et le MPU, et pour manipuler l'information à l'intérieur du MPU lui-même. Le chapitre suivant présentera, en détail, toutes les instructions du Z80. Le chapitre 5 montrera les techniques d'adressage, et le suivant les techniques d'échanges d'informations *avec l'extérieur* : les techniques d'entrée-sortie.

Dans ce chapitre, notre étude sera essentiellement liée à l'exercice concret de la programmation. En examinant des programmes de difficulté croissante, nous apprendrons les rôles respectifs des différentes instructions et des registres, et nous aurons l'occasion d'appliquer les concepts développés auparavant. Cependant, l'un deux, important, manquera à l'appel : le concept de technique d'adressage. En raison de son apparente complexité, il sera présenté, séparément, au chapitre 5.

Commençons donc à écrire quelques programmes pour le Z80, et d'abord des programmes arithmétiques. La carte des registres est présentée à la figure 3.0.

PROGRAMMES ARITHMÉTIQUES

Les programmes arithmétiques comprennent l'addition, la soustraction, la multiplication et la division. Nous travaillerons ici sur des entiers qui seront soit binaires positifs, soit représentés en complément à deux. Dans ce dernier cas, le bit le plus à gauche sera le bit de signe (voir, au chapitre 1, la description de la notation en complément à deux).

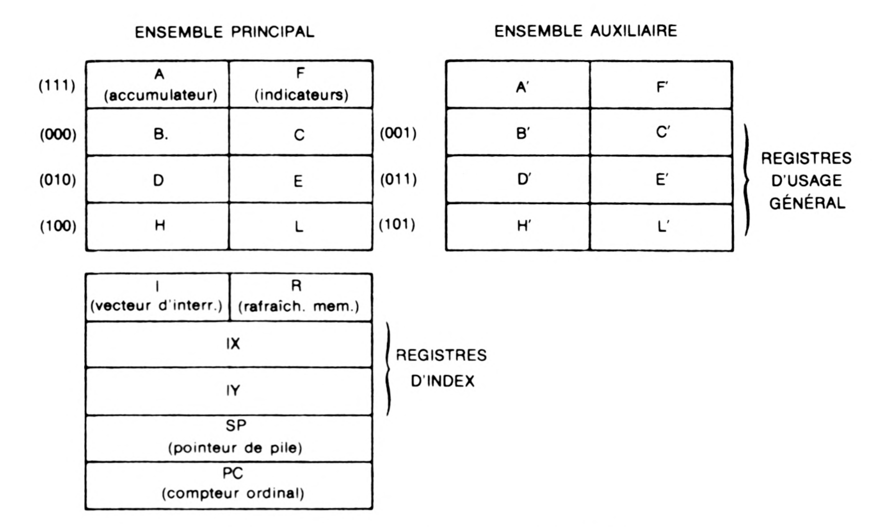


Figure 3.0. — Les registres du Z80

Addition sur 8 bits

Nous allons additionner deux opérandes de 8 bits, OP1 et OP2, respectivement rangés aux emplacements mémoire ADR1 et ADR2. Leur somme sera appelée RES, et rangée à l'adresse mémoire ADR3 (voir figure 3.1).

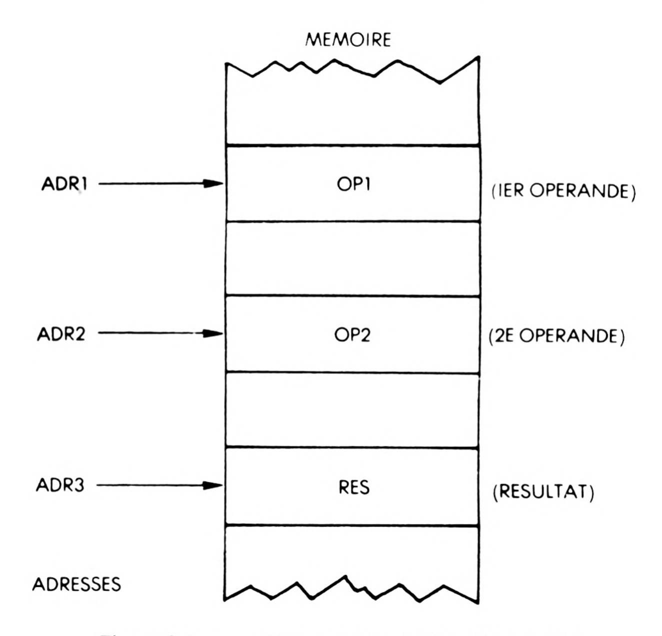


Figure 3.1. — Addition 8 bits RES = OP1 + OP2

Le programme effectuant cette addition est le suivant :

Instructions		Commentaires
LD	A, (ADR1)	CHARGER L'OPÉRANDE DANS A
LD	HL, ADR2	CHARGER L'ADRESSE DE OP2 DANS HL
ADD	A, (HL)	ADDITIONNER OP2 A OP1
LD	(ADR3), A	RANGER LE RÉSULTAT A L'ADRESSE ADR3

C'est là notre premier programme. Les instructions y sont écrites à gauche, et les commentaires à droite. Il comporte quatre instructions. Chaque ligne, appelée *instruction,* est exprimée sous forme symbolique, puis traduite par le programme *assembleur* en 1, 2, 3 ou 4 octets. Nous ne nous intéresserons pas ici à cette traduction, et nous contenterons de la représentation symbolique.

La première ligne signifie : charger le contenu de ADR1 dans l'accumulateur A. Nous reportant à la figure 3.1., nous voyons que le contenu de ADR1 représente le premier opérande OP1. Cette première instruction a donc pour résultat de transférer OP1 de la mémoire dans l'accumulateur (figure 3.2.). ADR1 est la représentation symbolique de l'adresse 16 bits réelle dans la mémoire. A un autre endroit du programme, ce symbole sera défini. Il pourrait l'être, par exemple, comme étant l'adresse « 100 ».

L'instruction de *chargement* va consister en une *opération de lecture* de l'adresse 100 (voir figure 3.2), dont le contenu sera transféré sur le bus de données, et déposé dans l'accumulateur.

Nous avons vu au chapitre précédent, que les opérations arithmétiques et logiques utilisent l'accumulateur comme l'un des opérandes sources (s'y reporter pour plus de détails). Pour additionner les deux valeurs OP1 et OP2, nous devrons d'abord charger OP1 dans l'accumulateur, avant d'être capables d'additionner le contenu de ce dernier, autrement dit OP1, à OP2. La partie droite de l'instruction est appelée *commentaire.* Elle est ignorée par l'assembleur au moment de la traduction, mais facilite la lecture du programme. Pour bien comprendre la nature des opérations mises en œuvre

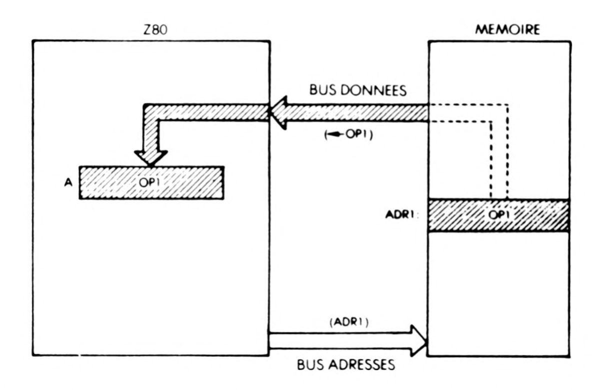


Figure 3.2. — LD A, (ADR1) : OP1 est chargé depuis la mémoire

par le programme, il est d'une importance primordiale de le commenter convenablement. Cela s'appelle *documenter* un programme.

Ici, le commentaire sert, à lui seul, d'explication. La valeur de OP1, logée à l'adresse ADR1, est chargée dans l'accumulateur A.

Le résultat de cette première instruction est illustré par la figure 3.2. La seconde instruction de notre programme est :

LD HL,ADR2

Elle signifie : charger les registres H et L avec la valeur ADR2. Pour lire le second opérande OP2, depuis la mémoire, nous devrons d'abord placer son adresse dans une des paires de registres. Ensuite, nous pourrons additionner le contenu de la case mémoire, dont l'adresse est dans HL, à l'accumulateur.

Reportons-nous à la figure 3.1. Nous voyons que le contenu de l'adresse mémoire ADR2 est notre second opérande, OP2. Le contenu de l'accumulateur est OP1. OP2 sera lu depuis la mémoire, et additionné à OP1 [figure 3.3.].

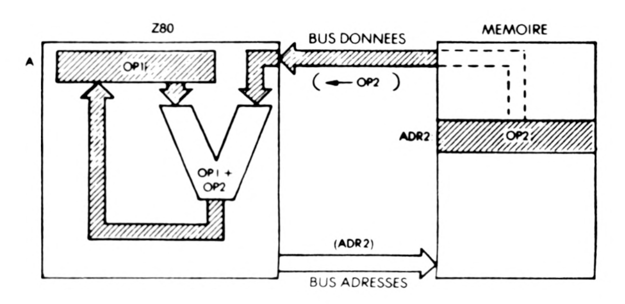


Figure 3.3. — ADD A, (HL)

La somme sera déposée dans l'accumulateur. Le lecteur doit se rappeler que, dans le cas du Z80, les résultats des opérations arithmétiques sont déposés dans l'accumulateur. Dans d'autres microprocesseurs, ces résultats peuvent être déposés dans d'autres registres, ou même, directement, en mémoire.

La somme de OP1 et OP2 se trouve maintenant dans l'accumulateur. Il reste simplement à transférer le contenu de l'accumulateur vers l'adresse ADR3, au moyen de la quatrième instruction de notre programme :

LD (ADR3), A

Cette instruction charge le contenu de A dans la case mémoire d'adresse ADR3. L'effet de cette dernière instruction est illustré par la figure 3.4.

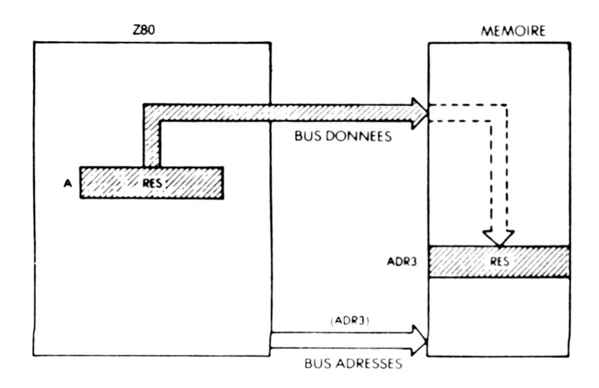


Figure 3.4. — LD (ADR3), A (Sauver l'accumulateur en mémoire)

Avant l'exécution de l'opération ADD, l'accumulateur contenait OP1 (voir figure 3.3.). Après, un nouveau résultat a été écrit : OP1 + OP2. Rappelons-nous que le contenu de n'importe quel registre de microprocesseur, comme celui de n'importe quelle case mémoire, n'est pas modifié par une opération de lecture. *Seule* une opération *d'écriture* en est capable. Dans notre exemple, le contenu des cases mémoires ADR1 et ADR2 reste le même tout au long du programme. Cependant, après l'instruction ADD, le contenu de l'accumulateur est modifié : la sortie de l'ALU y a été écrite, et le contenu précédent perdu.

Des adresses numériques réelles peuvent être utilisées à la place de ADR1, ADR2 et ADR3. Pour, cependant, continuer à utiliser ces adresses symboliques, il sera nécessaire d'utiliser des *pseudo-instructions,* précisant la valeur de ces adresses symboliques, de telle manière que l'assembleur puisse, pendant la traduction, leur substituer les valeurs réelles. Exemple de pseudo-instructions :

```
ADR1 = 100 H
ADR2 = 120 H
ADR3 = 200 H
```

Exercice 3.1 : *Refermez maintenant ce livre. Reportez-vous uniquement à la liste d'instructions, à la fin de ce livre. Ecrivez un programme qui additionne deux nombres situés aux emplacements mémoire LOC1 et LOC2. Déposez le résultat dans la case mémoire LOC3. Comparez ensuite votre programme à celui que nous venons de présenter.*

Addition sur 16 bits

L'addition sur 8 bits ne permet, avec du binaire absolu, que de travailler sur des nombres compris entre 0 et 255. Dans la plupart des applications, il est nécessaire d'additionner des nombres de 16 bits ou plus, c'est-à-dire

d'utiliser la *précision multiple.* Nous allons présenter des exemples arithmétiques sur des nombres de 16 bits, qui pourront facilement être étendus à 24, ou 32 bits, voire même davantage (mais toujours des multiples de 8). Nous supposerons que le premier opérande est rangé dans les cases mémoire ADR1 et ADR1 − 1. Dans la mesure où OP1 est maintenant un nombre de 16 bits, il a besoin de deux emplacements mémoire de 8 bits. De la même façon, OP2 sera logé aux adresses ADR2 et ADR2 − 1. Le résultat sera rangé aux adresses ADR3 et ADR3 − 1 [figure 3.5.].*H* désigne la partie haute (bits 8 à 15), et *L* la partie basse (bits 0 à 7). Cette convention dérive des deux mots anglais HIGH (haut) et LOW (bas).

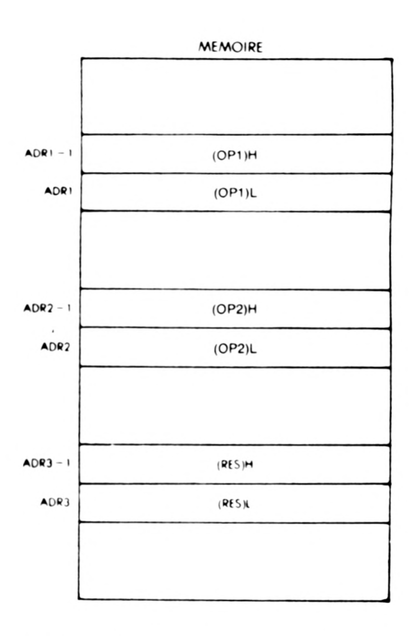


Figure 3.5. — Addition 16 bits — les opérandes

La logique de ce programme est exactement la même que celle du précédent. Nous additionnerons, en premier lieu, les parties basses des deux opérandes, puisque le microprocesseur ne peut additionner que 8 bits à la fois. Tout report généré par l'addition des deux octets sera, automatiquement, mémorisé dans le registre d'état C (carry).

Voici le programme :

```
LD   A, (ADR1)    CHARGER LA PARTIE BASSE DE ADR1
LD   HL, ADR2     ADRESSE DE LA PARTIE BASSE DE OP2
ADD  A, (HL)      ADDITIONNER LES PARTIES BASSES DE
                  OP1 ET OP2
```

```
LD   (ADR3), A         RANGER LA PARTIE BASSE DU RESUL-
                       TAT
LD   A, (ADR1 - 1)     CHARGER LA PARTIE HAUTE DE OP1
DEC  HL                ADRESSE PARTIE HAUTE DE OP2
ADC  A, (HL)           (OP1 + OP2) PARTIE HAUTE + REPORT
                       EVENTUEL
LD   (ADR3 - 1), A     RANGER PARTIE HAUTE DU RESULTAT
```

Les quatre premières instructions sont identiques à celles utilisées pour l'addition 8 bits, dans le programme précédent. Elles consistent à additionner les deux moitiés les moins significatives (bits 0 à 7) de OP1 et OP2. La somme, appelée RES, est rangée à l'adresse mémoire ADR3 (voir figure 3.5.).

Si l'addition des deux nombres génère un report, alors le bit « carry » (C) sera égal à 1 (il sera positionné). Si aucun report n'est généré, la valeur du bit de « carry » sera 0.

Les quatre instructions suivantes ressemblent beaucoup à celles utilisées dans le programme d'addition 8 bits. Cette fois, elles additionnent les deux octets de poids forts (bits 8 à 15) de OP1 et OP2, plus le report, s'il y en a un. Le résultat est rangé à l'adresse ADR3 − 1.

Après l'exécution de ce programme de 8 instructions, le résultat sur 16 bits est rangé dans les cases mémoire ADR3 et ADR3 − 1. A noter, cependant, la différence entre la deuxième et la première moitié du programme. L'instruction d'addition utilisée n'est pas la même. Dans la première moitié, il s'agit de la troisième instruction : l'instruction ADD, qui additionne les deux opérandes, sans se soucier de la valeur du bit de report. Dans la seconde moitié, il s'agit de ADC, qui additionne les deux opérandes et le bit de report ensemble. C'est une condition nécessaire à l'obtention du résultat correct. En effet, la première addition, effectuée sur les parties basses des deux opérandes, peut produire un report qui devra être pris en compte dans la seconde partie.

Une question vient naturellement à l'esprit : et si l'addition des deux parties hautes des opérandes produisait, aussi, un report ? Il y a deux réponses possibles. La première est de considérer qu'il y a eu une erreur. En effet, le programme a été conçu pour travailler sur des opérandes de 16 bits, et non de 17. La seconde est d'inclure des instructions supplémentaires, testant explicitement la présence d'un report à la fin du programme. Il y a là, pour le programmeur, un choix : le premier d'une longue série.

Remarque : nous avons supposé que la partie haute des opérandes se trouve « au-dessus » de la partie basse, c'est-à-dire à l'adresse mémoire immédiatement inférieure. Tel n'est pas toujours le cas. Les adresses sont notamment rangées en ordre inverse, dans le Z80 : partie basse d'abord, puis dans la case mémoire suivante, partie haute. De manière à utiliser la même convention pour les adresses et les données, il est recommandé de ranger ces dernières dans le même ordre : la partie basse au-dessus de la partie haute (voir figure 3.6.).

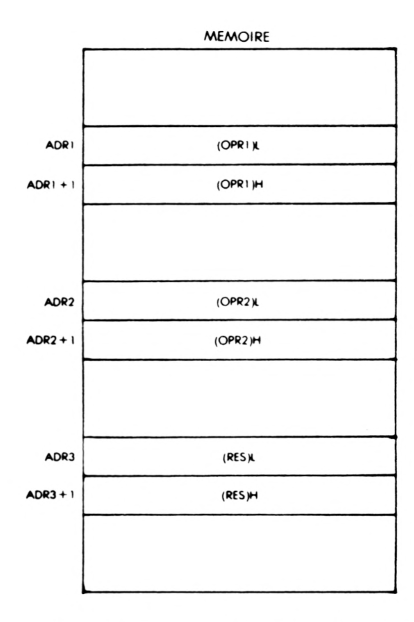


Figure 3.6. — Rangement des opérandes en ordre inversé

Lorsqu'on travaille sur des opérandes de plusieurs octets, il est nécessaire de fixer deux conventions :

— l'ordre dans lequel les données sont rangées en mémoire,
— l'octet qui devra être désigné par les pointeurs : le haut ou le bas.

Les exercices 3.2 et 3.3 vous aiderons à clarifier ce point.

Exercice 3.2 : *Réécrivez l'addition sur 16 bits, en utilisant l'implantation mémoire présentée à la figure 3.6.*

Exercice 3.3 : *Supposons maintenant que ADR1 ne pointe pas sur la partie basse de OP1 (comme sur les figures 3.5. ou 3.6.), mais sur la partie haute [voir figure 3.7.]. Réécrivez le programme, dans ce cas.*

C'est le programmeur, autrement dit vous-même, qui décide de la manière dont les nombres de 16 bits devront être rangés en mémoire (partie haute au-dessus ou partie basse au-dessus). Et si les adresses désignent la partie haute ou la partie basse du nombre. Cet autre choix, vous apprendrez à le faire en concevant des algorithmes ou des structures de données.

Les programmes précédents sont traditionnels : ils utilisent l'accumulateur. Une alternative est possible, pour l'addition 16 bits. Elle consiste à utiliser, non plus l'accumulateur, mais quelques unes des instructions 16 bits disponibles sur le Z80. Les opérandes sont supposés être rangés comme indiqué à la figure 3.6. Le programme est le suivant.

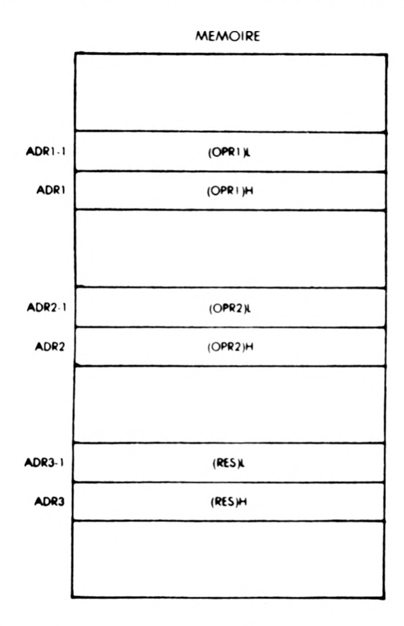


Figure 3.7. — Pointer sur l'octet de poids fort

```
LD   HL, (ADR1)     CHARGER OP1 DANS HL
LD   BC, (ADR2)     CHARGER OP2 DANS BC
ADD  HL, BC         ADDITION SUR 16 BITS
LD   (ADR3), HL     RANGER LE RESULTAT EN ADR3
```

Remarquez que ce programme est plus court que les versions précédentes. Il est aussi plus « élégant ». Dans une certaine mesure, le Z80 permet d'utiliser la paire de registres H et L en tant qu'accumulateur 16 bits.

Exercice 3.4 : *Utilisez les instructions 16 bits que nous venons d'introduire, pour écrire un programme d'addition sur les opérandes de 32 bits, en supposant que ces derniers sont rangés comme indiqué à la figure 3.8 (la réponse est présentée ci-dessous).*

Réponse :

```
LD   HL, (ADR1 − 1)
LD   BC, (ADR2 − 1)
ADD  HL, BC
LD   (ADR3 − 1), HL
LD   HL, (ADR1 − 3)
LD   BC, (ADR2 − 3)
ADC  HL, BC
LD   (ADR3 − 3), HL
```

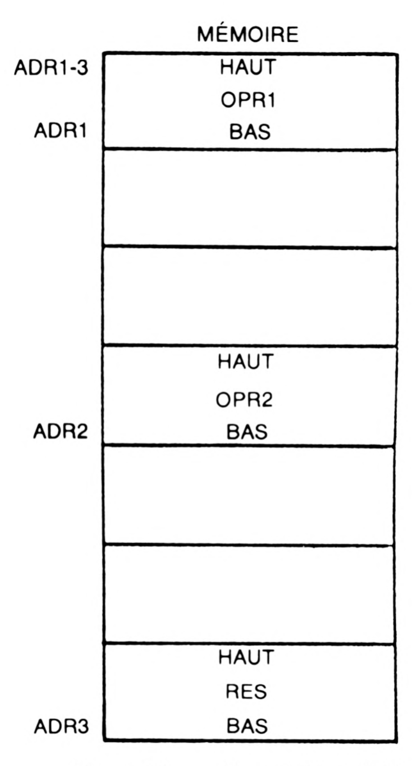


Figure 3.8. — Une addition 32 bits

Après l'addition binaire, voyons la soustraction.

Soustraire des nombres de 16 bits

Effectuer une soustraction sur 8 bits serait trop simple. Nous le garderons comme exercice, et passerons directement à la soustraction sur 16 bits. Comme d'habitude, nos deux nombres OP1 et OP2 sont rangés aux adresses ADR1 et ADR2. Nous supposerons que l'implantation mémoire est celle proposée à la figure 3.6. La soustraction recourra à l'instruction SBC, au lieu de ADD.

Exercice 3.5 : *Maintenant, écrivez le programme de soustraction*

Le programme est présenté ci-dessous, et le chemin emprunté par les données à la figure 3.9.

```
LD      HL, (ADR1)    OP1 dans HL
LD      DE, (ADR2)    OP2 dans DE
AND     A             MET L'INDICATEUR « CARRY » A 0
SBC     HL, DE        OP1 — OP2
LD      (ADR3), HL    RESULTAT EN ADR3
```

Ce programme ressemble beaucoup à celui de l'addition 16 bits. Cependant, l'ensemble d'instructions du Z80 présente deux additions possibles sur les paires de registres ADD et ADC, alors qu'il ne propose qu'une seule soustraction sur 16 bits : SBC.

De là deux changements :

- le premier est l'utilisation de SBC à la place de ADD.
- le second, le recours à l'instruction « AND A » pour mettre l'indicateur de « carry » (C) à 0, avant la soustraction. Cette instruction ne modifie pas le contenu de l'accumulateur.

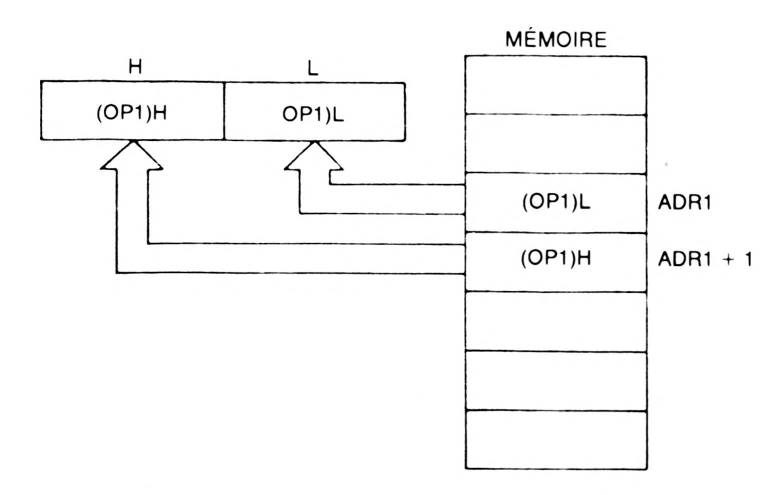


Figure 3.9. — Chargement 16 bits — LD HL, (ADR1)

Cette précaution est nécessaire du fait que le Z80 possède deux additions sur H et L (l'une tenant compte de l'indicateur C, et l'autre non), alors qu'il n'existe qu'une seule soustraction, SBC (de l'anglais SuBstract with Carry), qui elle tient compte de l'indicateur C. Dans cette mesure, l'indicateur doit être mis à 0 avant de commencer l'opération. C'est le rôle de l'instruction « AND A ».

Exercice 3.6 : *Réécrire le programme de soustraction sans utiliser les instructions sur 16 bits.*

Exercice 3.7 : *Ecrire le programme de soustraction pour des opérandes sur 8 bits.*

Il faut se souvenir que, dans le cas de l'arithmétique en complément à 2, la valeur finale de l'indicateur de report n'a pas de signification. Si un débordement a lieu à la suite de la soustraction, le bit indicateur de débordement (bit V) est positionné. Il peut être testé.

Les exemples précédents concernent des additions et des soustractions binaires. Toutefois, un autre type d'arithmétique est parfois nécessaire : l'arithmétique DCB (décimal codé binaire).

ARITHMÉTIQUE DCB

Addition DCB sur 8 bits

Le concept de l'arithmétique DCB a été présenté au chapitre 1. Rappelons ses caractéristiques. Cette arithmétique est essentiellement utilisée pour les applications de gestion, où il est impératif qu'aucun chiffre significatif des résultats ne sont perdu. Dans la notation DCB, un quartet de 4 bits sert à représenter un chiffre décimal (de 0 à 9). Par conséquent, tout octet de 8 bits peut être utilisé pour ranger deux chiffres DCB (cela s'appelle le *DCB compacté)*. Nous allons additionner deux octets, contenant chacun deux chiffres DCB.

Pour bien préciser le problème, nous aurons d'abord recours à des exemples numériques.

Additionnons « 01 » et « 02 ».

« 01 » est représenté par : 0000 0001
« 02 » est représenté par : 0000 0010
Le résultat est : 0000 0011

C'est là, justement, la représentation DCB de « 03 » (Si vous ne vous sentez pas à l'aise avec ce type de représentation, vous pouvez consulter la table de conversion DCB-binaire, à la fin du livre). Tout « marche » très simplement dans ce cas. Voyons maintenant un autre exemple.

« 08 » est représenté par 0000 1000
« 03 » est représenté par 0000 0011

Exercice 3.8 : *Qu'obtient-on en calculant la somme de ces deux nombres en représentation DCB ?*

Réponse : Si le résultat est « 0000 1011 », vous avez calculé la somme de 8 et de 3 en binaire. Soit 11 *en binaire*. Malheureusement, « 1011 » est une représentation DCB illégale. Il aurait fallu obtenir la représentation DCB de « 11 », c'est-à-dire 0001 0001 !

Le problème tient au fait que la représentation DCB n'utilise, pour coder les chiffres décimaux 0 à 9, que les dix premières combinaisons de 4 bits. Les six combinaisons restantes sont inutilisées, et le code illégal « 1011 » est l'une d'elles. En d'autres termes, lorsque la somme de deux chiffres binaires est supérieure à 9, il convient d'additionner 6 au résultat, pour sauter les six codes inutilisés.

Additionnons la représentation binaire de « 6 » à 1011 :

```
               1011    (résultat binaire illégal)
             + 0110    ( + 6)
             ------
résultat = 00010001
```

Il s'agit bien de la représentation DCB de « 11 » ! Le résultat est maintenant correct.

Cet exemple illustre l'une des difficultés fondamentales du mode de calcul DCB : la nécessité de compenser les six codes inutilisés. Une instruction spéciale, « DAA », appelée « ajustage décimal » (DAA, de l'anglais = Decimal Adjust) doit être utilisée pour corriger le résultat de l'addition binaire (en ajoutant 6 si le résultat est supérieur à 9).

Le problème suivant est illustré par le même exemple. Un report est ici généré du chiffre DCB de moindre poids (celui de droite) vers celui de poids fort (celui de gauche). Ce report interne doit être pris en compte et additionné au second chiffre DCB. L'instruction d'addition réalise automatiquement cette opération. Cependant, il est souvent utile de pouvoir détecter ce report du bit 3 vers le bit 4 (le report « au milieu »). L'indicateur H (de l'anglais Half carry) y pourvoit.

Voici, par exemple, un programme d'addition des deux nombres DCB « 11 » et « 22 » :

```
LD   A, 11H        CHARGER LA VALEUR DCB « 11 »
ADD A, 22H         Y AJOUTER LA VALEUR DCB « 22 »
DAA                CORRECTION DECIMALE
LD   (ADR), A      RANGEMENT DU RESULTAT
```

Un nouveau symbole, « H », est utilisé ici. Placé dans la partie opérande de l'instruction, il signifie que la donnée suivante est exprimée en notation hexadécimale. Les représentations hexadécimale et DCB sont identiques pour les chiffres de 0 à 9. Ici, il s'agit d'additionner les deux littéraux (ou constantes) « 11 » et « 22 ». Le résultat est rangé à l'adresse ADR. Lorsque, comme c'est le cas ici, l'opérande fait partie de l'instruction, nous dirons que l'adressage est *immédiat* (les différents modes d'adressage seront décrits, en détail, au chapitre 5). Si le résultat est rangé à une adresse spécifiée dans l'instruction [exemple : LD (ADR), A,], l'adressage est dit *absolu*.

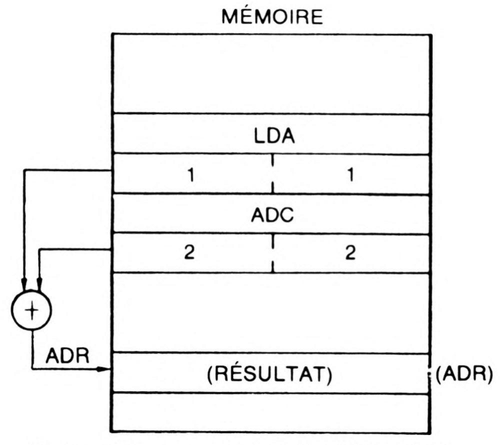


Figure 3.10. — rangement de chiffres DCB

Ce programme est analogue à celui de l'addition binaire sur 8 bits, mais utilise une nouvelle instruction : « DAA ». Son rôle sera illustré par un exemple. Additionnons d'abord « 11 » et « 22 » en DCB :

```
    0001 0001   (11)
  + 0010 0010   (22)
    ---------
  = 0011 0011   (33)
     ~~~  ~~~
      3    3
```

Le résultat est correct, selon les règles de l'addition binaire.

Additionnons maintenant « 22 » et « 39 », selon ces mêmes règles :

```
    0010 0010   (22)
  + 0011 1001   (39)
    ---------
  = 0101 1011
     ~~~  ~~~
      5    ?
```

« 1011 » est un code DCB illégal, parce que l'arithmétique DCB n'utilise que les dix premiers codes possibles avec 4 bits, en « sautant » les six suivants. Il faut donc faire la même chose, et additionner 6 au résultat :

```
    0101 1011   (résultat binaire)
  +      0110   ( 6)
    ---------
  = 0110 0001   (61)
     ~~~  ~~~
      6    1
```

Le résultat DCB est correct.

***Exercice 3.9 :** Est-il possible de déplacer l'instruction DAA pour la mettre derrière l'instruction LD (ADR), A ?*

Soustraction DCB

La soustraction DCB est, en apparence, complexe. Elle nécessite que le *complément à dix* du nombre soit additionné de la même manière que le complément à deux dans le cas d'une soustraction binaire. Le complément à dix s'obtient en calculant le complément à neuf, et en ajoutant 1 au résultat. Cela demande, en général, trois ou quatre opérations sur un microprocesseur ordinaire. Mais le Z80 est équipé d'une instruction DAA très puissante, qui simplifie le programme. Elle corrige automatiquement la valeur du résultat contenu dans l'accumulateur, en fonction de la valeur des indicateurs C et H juste avant son exécution. (Voir le chapitre suivant pour plus de détails sur l'instruction DAA).

Addition DCB 16 bits

Elle s'effectue aussi simplement qu'en arithmétique binaire.
Voici un exemple de programme :

```
LD   A, (ADR1)      CHARGER LA PARTIE BASSE DE OP1
                    DANS A
LD   HL, ADR2       CHARGER ADR2 DANS HL
ADD A, (HL)         PARTIE BASSE DE (OP1 + OP2)
DAA                 CORRECTION DECIMALE
LD   (ADR3), A      RANGER LA PARTIE BASSE DU
                    RESULTAT
LD   A, (ADR1 + 1)  PARTIE HAUTE DE OP1 DANS A
INC  HL             POINTER SUR PARTIE HAUTE DE OP2
ADC A, (HL)         PARTIE HAUTE DE (OP1 + OP2)
                    + REPORT ÉVENTUEL
DAA                 CORRECTION DECIMALE
LD   (ADR3 + 1)     RANGER LA PARTIE HAUTE DU
                    RESULTAT
```

Soustraction DCB compacté

L'addition et la soustraction DCB élémentaires viennent d'être décrites. Cependant, en pratique, les nombres DCB peuvent comporter un nombre quelconque de chiffres. Pour simplifier notre exemple de soustraction en

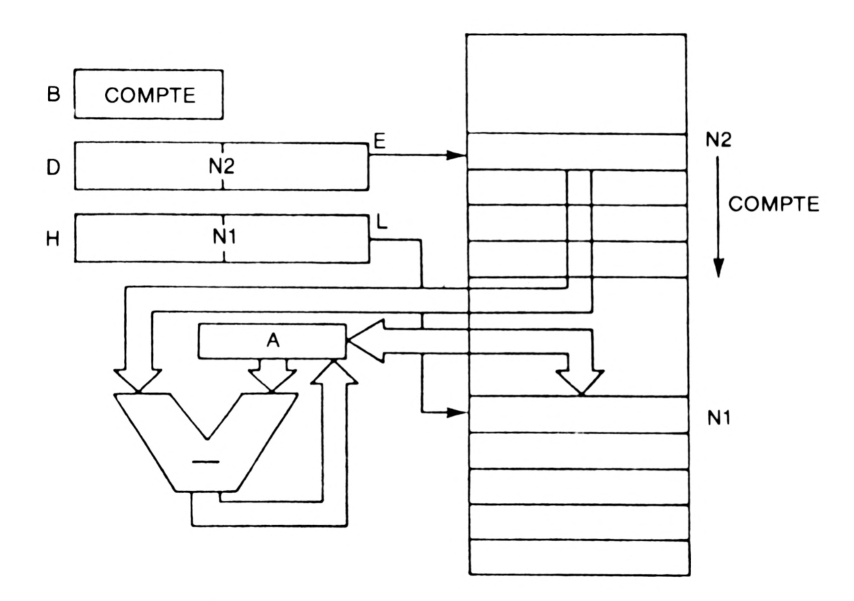


Figure 3.11. — Soustraction DCB compacté : N1←N2 — N1

format DCB compacté, nous supposerons que les deux nombres N1 et N2 disposent du même nombre d'octets DCB : soit COMPTE. Les registres et l'utilisation de la mémoire sont présentés à la figure 3.11.

Voici le programme :

```
DCBSUB      LD    B, COMPTE
            LD    DE, N2
            LD    HL, N1
            AND   A            EFFACE UNE INDICATION
                               EVENTUELLE DE REPORT
SOUSTRAIRE  LD    A, (DE)      UN OCTET DE N2
            SBC   A, (HL)      MOINS UN OCTET DE N1
            DAA
            LD    (HL), A      RANGER LE RESULTAT
            INC   DE
            INC   HL
            DJNZ  SOUSTRAIRE   B←B — 1 ; BOUCLER
                               JUSQU'A CE QUE B = 0
```

N1 et N2 représentent les adresses où les deux nombres DCB sont rangés. Elles sont chargées dans les paires de registres HL et DE :

```
DCBSUB      LD    B, COMPTE
            LD    DE, N2
            LD    HL, N1
```

En anticipant sur la première soustraction, nous enlèverons une indication éventuelle de report. On sait que cette opération est nécessaire dans un certain nombre de cas semblables. Ici, nous utiliserons, par exemple :

```
AND A
```

Le premier octet de N2 est chargé dans l'accumulateur, et le premier octet de N1 lui est soustrait. L'instruction DAA permet ensuite d'obtenir la valeur DCB correcte :

```
SOUSTRAIRE  LD    A, (DE)
            SBC   A, (HL)
            DAA
```

Le résultat est rangé dans N1 :

```
LD    (HL), A
```

Finalement, les pointeurs sur l'octet courant sont incrémentés.

```
INC   DE
INC   HL
```

Le compteur est décrémenté, et on boucle tant qu'il n'a pas atteint 0 :

```
DJNZ   SOUSTRAIRE
```

L'instruction DJNZ est propre au Z80. Elle permet, en une seule instruction de décrémenter le contenu du registre B, et d'effectuer un branchement tant que B n'a pas atteint la valeur 0.

Exercice 3.10 : *Comparer le programme précédent au programme d'addition binaire sur 16 bits. Quelles sont les différences ?*

Exercice 3.11 : *Pourriez-vous échanger les rôles de DE et de HL ? (attention : soyez prudents avec SBC).*

Exercice 3.12 : *Ecrivez un programme de soustraction DCB sur 16 bits.*

Les indicateurs et le mode DCB

En arithmétique DCB, une indication de report à la suite d'une addition signifie que le résultat dépasse 99. Ce n'est pas la même chose qu'en complément à deux, puisque les chiffres DCB sont représentés en binaire pur. Réciproquement, la présence de l'indicateur de report, après une soustraction, indique une retenue.

Les types d'instructions

Nous avons maintenant utilisé deux types d'instructions du microprocesseur.

En premier lieu LD, qui permet le chargement de l'accumulateur à partir du contenu d'une adresse mémoire, ou au contraire, le rangement du contenu de l'accumulateur à une adresse mémoire. Il s'agit d'une instruction de *transfert de données.*

Ensuite, nous avons utilisé des instructions *arithmétiques,* telles que ADD, SUB, ADC et SBC, permettant d'effectuer des additions et des soustractions. D'autres instructions utilisant l'ALU seront introduites ultérieurement.

Bien d'autres instructions, que nous n'avons pas encore utilisées, existent sur ce microprocesseur. En particulier les instructions de « saut », qui modifient l'ordre d'exécution des instructions du programme. Ce nouveau type sera introduit dans notre prochain exemple. Remarquons que les instructions de saut sont souvent dites de « branchement » dans les situations conditionnelles, c'est-à-dire lorsqu'un choix logique est à faire. Le mot « branchement » présente une analogie avec un arbre, et indique une « fourche » (d'un côté ou de l'autre) dans la représentation du programme.

MULTIPLICATION

Examinons un problème arithmétique plus complexe : la multiplication des nombres binaires. Pour introduire l'algorithme correspondant, nous

commencerons par examiner une multiplication décimale, avec laquelle nous sommes familiarisés. Multiplions donc 12 par 23 :

```
       12   (multiplicande)
  ×    23   (multiplicateur)
  -------
       36   (produit partiel)
  +   24
  -------
  =   276   (résultat final)
```

Il convient de multiplier d'abord le multiplicande par le chiffre placé le plus à droite du multiplicateur. Autrement dit : « 12 » × « 3 «. Le produit partiel obtenu est « 36 ». Puis, le multiplicande est multiplié par le chiffre suivant du multiplicateur. Soit, ici : « 12 » × « 2 ». Le produit partiel « 24 » est ajouté au produit partiel précédent.

Une opération supplémentaire est à effectuer. 24 est d'abord *décalé à gauche* d'une position. De la même manière, nous pourrions dire que « 36 » est décalé d'une position vers la droite, avant l'addition.

Les deux nombres, correctement décalés, sont ensuite ajoutés. Leur somme est égale à 276. C'est simple. La multiplication binaire s'effectue exactement de la même façon. Exemple : multiplions 5 par 3.

```
(5)             101   (multiplicande,MPD)
(3)       ×     011   (multiplicateur, MPR)
          ---------
                101   (produit partiel, PP)
               101
              000
          ---------
(15)      =   01111   (résultat, RES)
```

Tel est exactement le procédé utilisé pour effectuer notre multiplication. La représentation formelle de cet algorithme apparaît à la figure 3.12. C'est l'ordinogramme de notre algorithme : notre premier ordinogramme. Examinons-le de plus près.

Il s'agit d'une représentation symbolique de l'algorithme que nous venons de présenter. Chaque rectangle représente un ordre à exécuter, traduit par une ou plusieurs instructions du programme. Chaque losange est un test à effectuer. Il constituera un point de branchement du programme. Si le test est probant, nous irons nous brancher à l'endroit indiqué. S'il ne l'est pas, nous nous brancherons ailleurs. Le concept de branchement sera expliqué plus loin, dans le programme lui-même.

Le lecteur devrait maintenant se pencher sur cet ordinogramme, pour se convaincre qu'il représente bien l'algorithme présenté. Remarquons qu'une flèche remonte du losange du bas vers celui du haut. Pourquoi ? Parce que la même partie de l'ordinogramme sera exécutée huit fois : une pour chaque bit du multiplicateur. Une telle situation où l'exécution repart du même point s'appelle, pour des raisons évidentes, une *boucle de programme.*

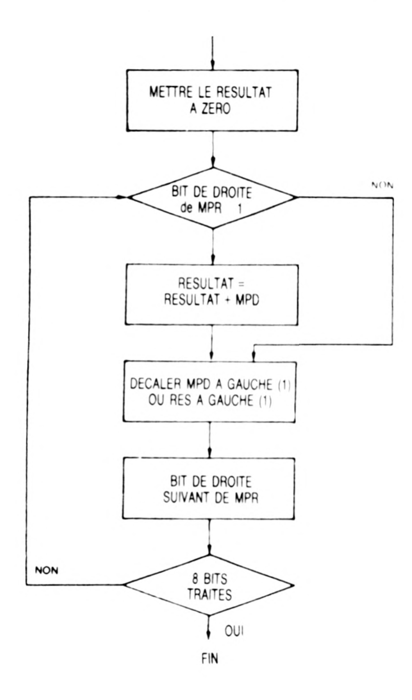


Figure 3.12. — Algorithme de multiplication de base-ordinogramme

***Exercice 3.13 :** Multiplier « 4 » par « 7 » en binaire, en suivant l'ordinogramme. Vérifiez que le résultat obtenu est bien « 28 ». Si tel n'est pas le cas, essayez à nouveau. Vous n'essaierez de traduire cet ordinogramme en programme que lorsque vous serez parvenus au bon résultat.*

Multiplication 8 bits par 8 bits

Traduisons maintenant cet ordinogramme en programme à l'intention du Z80. Le programme complet est présenté à la figure 3.13. Nous l'étudierons en détail. Comme nous l'avons souligné au chapitre 1, la programmation consiste ici à traduire un ordinogramme [figure 3.12] en programme [figure 3.13].

Chaque boîte de l'ordinogramme sera représentée par une ou plusieurs instructions.

Nous supposerons que MPD et MPR sont déjà pourvus d'une valeur.

```
MULT88  LD   BC, (MPRAD)   MULTIPLICATEUR DANS C
        LD   B, 8          B SERA UTILISE COMME
                           COMPTEUR
        LD   DE, (MPDAD)   MULTIPLICANDE DANS E
        LD   D, 0          INITIALISE D A 0
        LD   HL, 0         INITIALISE LE RESULTAT A 0
MULT    SRL  C             DECALE UN BIT DU MULTI-
                           PLICATEUR DANS L'INDICA-
                           TEUR DE REPORT
        JR   NC, NOADD     TESTE L'INDICATEUR DE
                           REPORT
        ADD  HL, DE        ADDITIONNE MPD AU
                           RESULTAT
NOADD   SLA  E             DECALE MPD VERS LA
                           GAUCHE
        RL   D             EN SAUVANT LE BIT SORTI
                           DANS D
        DEC  B             DECREMENTE LE
                           COMPTEUR DE BITS
        JP   NZ, MULT      ET RECOMMENCE TANT
                           QU'IL EST ≠ DE 0
        LD   (RESAD), HL   RANGE LE RESULTAT
```

Figure 3.13. — Programme de multiplication 8 par 8

La première boîte de l'ordinogramme est la *boîte d'initialisation,* indispensable pour mettre à 0 un certain nombre de registres, ou de cases mémoires, que le programme sera amené à utiliser. Les registres utilisés par le programme de multiplication sont présentés à la figure 3.14.

La multiplication recourt à trois paires de registres du Z80. Le multiplicateur sur 8 bits est supposé se trouver à l'adresse MPRAD, et le multiplicande à l'adresse MPDAD. Ils seront chargés, respectivement, dans les registres C et E (voir figure 3.14). Le registre B est utilisé comme compteur.

Les registres D et E serviront à contenir le multiplicande, puisque celui-ci sera décalé, à chaque fois, d'une position vers la gauche.

Remarque : bien que C et E aient seuls besoin d'être initialement chargés, nous avons utilisé des chargements sur 16 bits, B et D ont donc été chargés par la même occasion. B et D ont ensuite été « nettoyés », en étant initialisés respectivement à « 8 » et « 0 ».

Remarquons aussi que le résultat d'une multiplication 8 × 8 bits peut exiger 16 bits, parce que $2^8 \times 2^8 = 2^{16}$. Deux registres doivent donc être utilisés pour exprimer le résultat : ici, les registres H et L (cf. figure 3.14).

La première étape consiste à charger les registres B, C et E avec les valeurs convenables, et à initialiser le résultat (qui servira à accumuler les

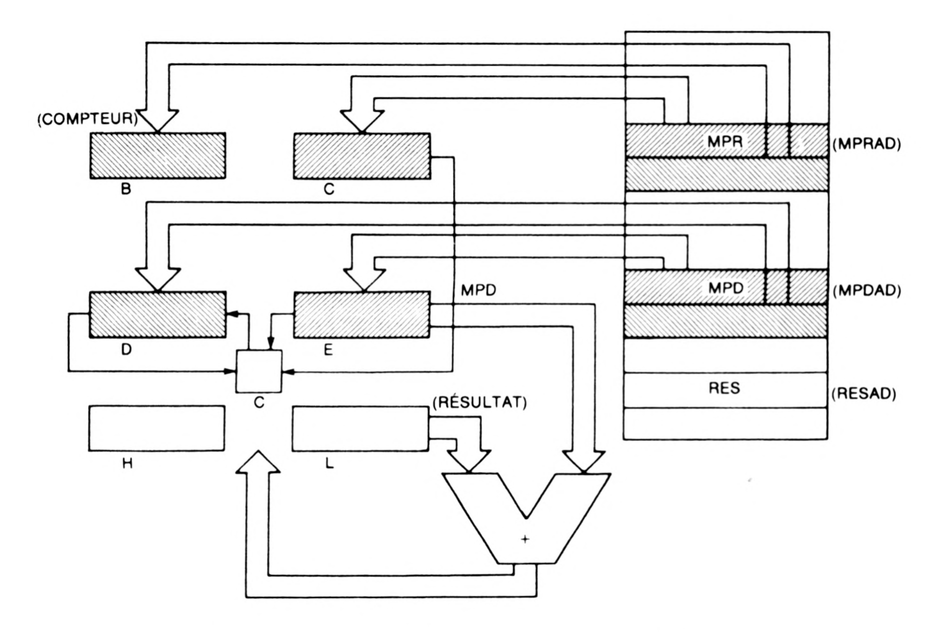


Figure 3.14. — Multiplication 8 × 8 — Les registres

produits partiels) avec la valeur « 0 », comme indiqué dans l'ordinogramme de la figure 3.12. Les instructions correspondantes sont les suivantes :

```
MULT 88 LD    BC, (MPRAD)
        LD    B, 8
        LD    DE, (MPDAD)
        LD    D, 0
        LD    HL, 0
```

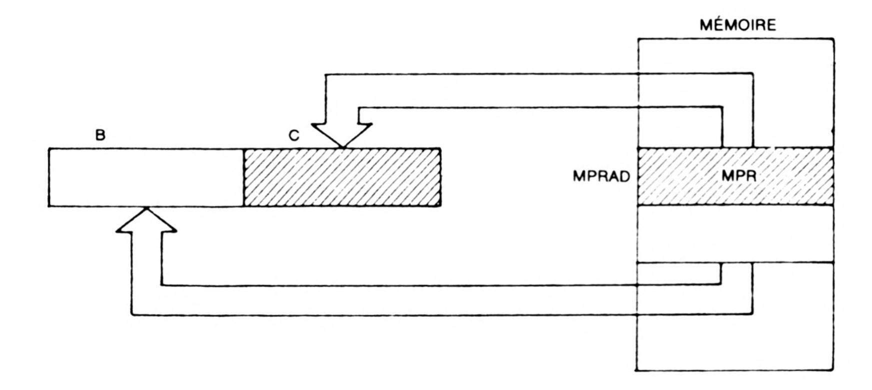


Figure 3.15. — LD BC, (MPRAD)

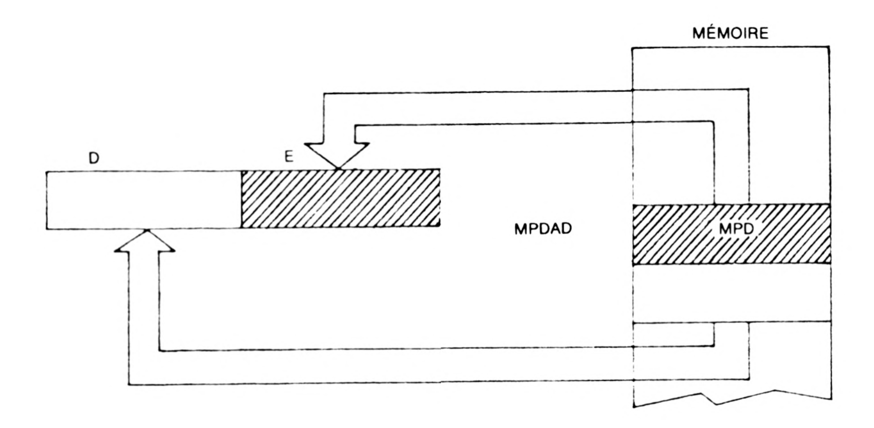


Figure 3.16. — LD DE, (MPDAD)

Les trois premières instructions chargent MPR dans la paire BC, la valeur « 8 » dans le registre B, et MPD dans la paire DE. Comme MPR et MPD sont des mots de 8 bits, ils sont, en fait, respectivement chargés dans C et E, alors que les mots situés aux adresses MPRAD + 1 et MPDAD + 1 le sont dans B et D. A la vérité, ces mots ne nous intéressent pas, et nous les remplacerons par les valeurs appropriées (cf. figures 3.15 et 3.16). L'instruction suivante met le contenu de D à 0.

Dans ce programme de multiplication, le multiplicande sera décalé d'une position vers la gauche, avant d'être additionné au résultat (rappelons que, comme indiqué dans la quatrième boîte de l'ordinogramme de la figure 3.12, nous aurions pu, à la place, décaler le résultat d'une position vers la droite). Le multiplicande MPD sera donc décalé vers la gauche, dans le registre D, à chaque étape. Au départ, le registre D doit être initialisé à 0, par l'intermédiaire de la quatrième instruction. Finalement, la cinquième instruction initialise H et L à 0.

En nous reportant à l'ordinogramme de la figure 3.12, nous voyons que l'étape suivante consiste à tester le bit de faible poids (le bit le plus à droite) du multiplicateur MPR. Si ce bit est à 1, il faut ajouter la valeur de MPD au résultat partiel, avec les trois instructions suivantes :

```
MULT    SRL    C
        JR     NC, NOADD
        ADD    HL, DE
```

Premier problème : le test du bit à l'extrême-droite du multiplicateur situé dans le registre C. Nous aurions pu utiliser l'instruction BIT du Z80, qui permet de tester n'importe quel bit de n'importe quel registre. Mais il nous a paru utile de construire un programme utilisant une boucle aussi simple que possible. Avec l'instruction BIT, nous aurions d'abord dû tester le bit 0, puis le bit 1, et ainsi de suite jusqu'au bit 7. D'où la nécessité à

chaque fois, d'une instruction nouvelle. Une simple boucle n'aurait pas fait l'affaire. C'est pour raccourcir le programme que nous avons utilisé une autre instruction : une instruction de *décalage*.
Remarque : il y a bien une manière d'utiliser simultanément l'instruction BIT et une boucle, mais sa mise en œuvre aurait entraîné une modification du programme par lui-même, ce qu'il faut éviter, à tout prix, en pratique.

SRL est une nouvelle instruction du type arithmétique et logique. Elle signifie : « décaler logiquement à droite » (de l'anglais Shift Right Logical). *Un décalage logique* est caractérisé par le fait que chaque bit est décalé d'une position vers la droite en « rentrant » un 0 dans le bit 7. A l'inverse du *décalage arithmétique,* où la nouvelle valeur du bit 7 est égale à l'ancienne. Les différents types d'instructions de décalage seront décrits au prochain chapitre.

L'effet de l'instruction SRL C est figuré sur la figure 3.4 par une flèche qui sort de la droite du registre C et entre dans le carré représentant l'indicateur de report (C) (C vient de l'anglais Carry, qui signifie report). A ce point, le bit le plus à droite de MPR se trouve dans l'indicateur de report C, où il peut être testé.

L'instruction suivante, « JR NC, NOADD » est une instruction de *saut.* Elle implique de se brancher à l'adresse NOADD, si l'indicateur de report n'est pas positionné (de l'anglais Jump when No Carry). Si le contenu de ce dernier est 0 (C = 0, pas de report), le programme saute à l'adresse NOADD. S'il est « 1 » (ne pas confondre l'indicateur C avec le registre C), alors le test échoue, le branchement n'a pas lieu et le programme continue en séquence, exécutant alors l'instruction « ADD HL, DE ».

Cette instruction signifie : additionner les contenus respectifs des paires DE et HL, et mettre le résultat dans HL. Puisque E contient le multiplicande MPD (voir figure 3.14), cela revient à additionner le multiplicande au résultat partiel.

A ce point, que le multiplicande ait ou non été ajouté au résultat partiel, il convient de le décaler d'une position vers la gauche (quatrième boîte de l'ordinogramme de la figure 3.12) au moyen de :

```
NOADD    SLA E
```

SLA signifie « décalage arithmétique vers la gauche (de l'anglais Shift Left Arithmetic) ». Nous avons vu plus haut qu'il existe deux types de décalages : les décalages arithmétiques et les décalages logiques. SLA est arithmétique. Dans le cas du décalage vers la gauche, le bit entrant à droite du registre (bit le moins significatif) est un « 0 » (tout à fait comme dans le cas du SRL ci-dessus).

La figure 3.14 montre que notre objectif est aussi de déplacer le bit le plus significatif (le MSB, de l'anglais Most Significant Bit) de E dans le registre D (regarder, sur la figure, la flèche allant de E vers D). Mais nulle instruction ne permet de décaler, en une seule fois, tous les bits d'une paire de registres. Lorsque E est décalé, son MSB « tombe » dans l'indicateur C.

Il faut l'y récupérer pour le rentrer dans D, par la droite. L'instruction suivante s'en charge :

RL D

RL est encore un nouveau type d'instruction. Elle signifie : effectuer la rotation vers la gauche (RL = Rotate Left). Lors d'une *rotation,* à la différence d'un décalage, le bit entrant dans le registre est celui contenu dans l'indicateur C juste avant l'opération (voir figure 3.17). C'est exactement ce que nous voulons. Le contenu de l'indicateur C est entré à droite du registre D, et le bit le plus à gauche du registre E a ainsi été déplacé dans le bit le plus à droite du registre D.

Cette séquence de deux instructions est illustrée sur la figure 3.18. Remarquons que le bit marqué X, à l'extrême-gauche de E, est d'abord transféré dans l'indicateur C, puis dans la position située la plus à droite de D. Il a ainsi été, effectivement, décalé de E vers D.

Revenons à l'ordinogramme de la figure 3.12. Pour aller chercher le bit suivant de MPR, et tester si l'opération a déjà été réalisée huit fois, il est nécessaire de décrémenter le compteur contenu dans le registre B (cf. figure 3.14). Ce dernier est décrémenté par l'instruction :

DEC B

dont la fonction est évidente.

Enfin, il faudra tester la valeur de l'indicateur Z pour s'assurer qu'après l'opération, le compteur a bien atteint la valeur 0. Le lecteur se souvient, sans doute, que l'indicateur Z (Zéro) révèle si l'instruction arithmétique qui vient d'être exécutée a produit un résultat égal à 0. A noter, cependant, que les instructions de décrémentation de quantités sur 16 bits (DEC HL, DEC BC, DEC DE, DEC IX et DEC IY) n'affectent pas l'indicateur Z. Si le compteur n'a pas atteint la valeur « 0 », l'opération n'est pas déterminée, et la boucle est à nouveau exécutée, au moyen de l'instruction suivante :

JP NZ, MULT

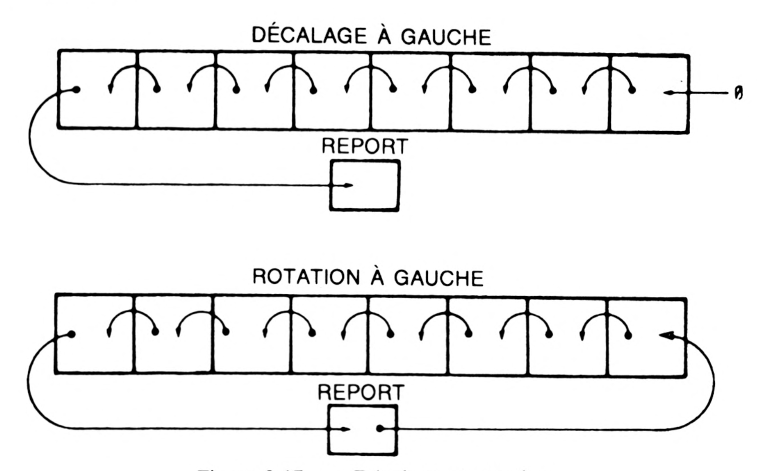


Figure 3.17. — Décalage et rotation

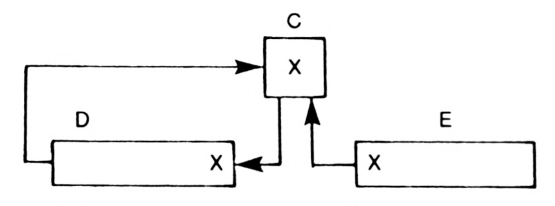


Figure 3.18. — Décalage de E vers D

Il s'agit d'une instruction de saut qui spécifie que, si l'indicateur Z n'est pas positionné (condition NZ = Non Zéro), un saut doit être effectué vers l'adresse MULT. La boucle du programme sera exécutée, de manière répétitive, jusqu'à ce que B atteigne la valeur 0. Lorsqu'il l'atteindra, en étant décrémenté de 1, le bit Z sera positionné, et le test JP NZ échouera. L'instruction suivante de la séquence sera alors exécutée :

LD (RESAD), HL

Cette instruction range, tout simplement, le contenu de HL, autrement dit le résultat de la multiplication, à l'adresse RESAD. Remarquez qu'elle transfère simultanément le contenu des registres H et L à deux emplacements mémoire consécutifs : L à l'adresse RESAD et H à l'adresse RESAD + 1. Cette instruction permet donc de ranger 16 bits d'un coup en mémoire.

Exercice 3.14 : *Réécrire le même programme de multiplication en utilisant l'instruction BIT (décrite au chapitre suivant) à la place de l'instruction SRL C. Quels sont ici les désavantages ?*

Essayons, si possible, d'améliorer notre programme :

Exercice 3.15 : *Peut-on substituer l'instruction JR à l'instruction JP à la fin du programme ? Si oui, quel est l'avantage ?*

Exercice 3.16 : *Peut-on utiliser l'instruction DJNZ pour raccourcir encore le programme ?*

Exercice 3.17 : *Examinons les deux instructions : LD D, 0 et LD HL, 0 au début du programme. Peut-on leur substituer la séquence suivante :*

```
XOR  A
LD   D, A
LH   H, A
LD   L, A
```

Si oui, quel est l'impact de cette substitution sur la taille du programme (nombre d'octets) et sur sa vitesse ?

Soulignons enfin que, dans la plupart des cas, le programme que nous venons de développer sera, en fait, un sous-programme, et que sa dernière

instruction sera un RET (retour). Le mécanisme des sous-programmes sera expliqué plus loin dans ce chapitre.

Auto-test important

Nous venons de rencontrer notre premier programme important. Il contient de nombreux types d'instructions différents : instructions de transfert (LD), opérations arithmétiques (ADD), opérations logiques (SRL, SLA, RL), opérations de saut (JR, JP), etc. Il contient aussi une boucle de programme : les sept instructions commençant à l'adresse MULT sont exécutées de manière répétitive. Pour comprendre la programmation, il est essentiel de comprendre parfaitement, et dans ses moindres détails, le fonctionnement d'un tel programme. Un exercice important vous est proposé en ce sens. Nous vous recommandons instamment d'y répondre complètement et correctement avant de continuer. Il s'agira de la seule

ETIQUETTE	INSTRUCTION	B	C	C (REPORT)	D	E	H	L

Figure 3.19. — Tableau pour l'exercice de multiplication

preuve réelle que les concepts présentés ont bien été assimilés. Si votre résultat est correct, cela signifie que vous avez vraiment compris les mécanismes par lesquels les instructions manipulent l'information dans le microprocesseur, la transfèrent entre les registres et la mémoire, et enfin la traitent. Dans le cas contraire il est probable que vous éprouverez des difficultés à écrire plus tard vos propres programmes. L'apprentissage de la programmation passe par une pratique personnelle. Arrêtez-vous donc maintenant, prenez une feuille de papier (ou utilisez le tableau 3.19) et faites l'exercice suivant :

Exercice 3.18 : *Chaque fois qu'un programme vient d'être écrit, il convient d'en vérifier, « à la main », le fonctionnement, de manière à s'assurer qu'il produit les bons résultats. C'est ce que nous allons faire ; le but de cet exercice est de remplir, complètement et précisément, le tableau 3.19.*

Vous pouvez utiliser directement ce tableau ou en faire une copie. Il est nécessaire de déterminer, du début à la fin, le contenu de chaque registre du Z80 concerné par l'exécution de chaque instruction du programme. Tous les registres utilisés par le programme de la figure 3.13 apparaissent sur le tableau 3.19. De gauche à droite, nous trouvons les registres B et C, l'indicateur C, les registres D et E, et enfin les registres H et L. Sur la partie gauche du tableau, il faudra porter éventuellement les étiquettes du programme, puis les instructions exécutées. Sur la droite, figurera le contenu des registres, après l'exécution de l'instruction. Chaque fois que le contenu d'un registre restera inconnu, vous mettrez des tirets.

Commençons, ensemble, à remplir le tableau. Vous continuerez ensuite par vous-même. Voici la première ligne :

ETIQUETTE	INSTRUCTION	B	C	C (REPORT)	D	E	H	L
		--	--	-	--	--	--	--
MULT88	LD BC$_1$, (0200)	--	03	-	--	--	--	--

Figure 3.20. — Multiplication : après une instruction

Nous supposerons ici qu'il s'agit de multiplier « 3 » (MPR) par « 5 » (MPD).

La première instruction à exécuter est « LD BC, (MPRAD) ». Le contenu de la case mémoire MPRAD est chargé dans les registres B et C. Nous supposerons que MPR est égal à 3, soit « 00000011 ». Après exécution de cette instruction, le contenu de C devient 3. Remarquons que, du même coup, le contenu de la case mémoire MPRAD + 1 a été chargé dans B. Cela a peu d'importance, puisque dès l'instruction suivante nous y mettrons la valeur « 8 » (cf. figure 3.21). A ce moment, les contenus de D, E, H et L sont encore inconnus, ce qui est indiqué par des tirets. L'instruction LD ne touchant pas à l'indicateur C, son contenu est également indéfini.

ETIQUETTE	INSTRUCTION	B	C	C (REPORT)	D	E	H	L
		--	--	-	--	--	--	--
MULT 88	LD BC, (0200)	--	03	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--

Figure 3.21. — Multiplication : après deux instructions

La situation après exécution des cinq premières instructions du programme (juste avant MULT) est présentée ci-dessous.

ETIQUETTE	INSTRUCTION	B	C	C (REPORT)	D	E	H	L
		--	--	-	--	--	--	--
MULT 88	LD BC,(0200)	--	03	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--
	LD DE,(0202)	08	03	-	00	05	--	--
	LD D, 00	08	03	-	00	05	--	--
	LD HL,0000	08	03	-	00	05	00	00

Figure 3.22. — Multiplication : après cinq instructions

ETIQUETTE	INSTRUCTION	B	C	C (REPORT)	D	E	H	L
		--	--	-	--	--	--	--
MULT 88	LD BC,(0200)	--	03	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--
	LD DE,(0202)	08	03	-	00	05	--	--
	LD D, 00	08	03	-	00	05	--	--
	LD HL,0000	08	03	-	00	05	00	00
MULT	SRL C	08	01	1	00	05	00	00
	JR NC,0114	08	01	1	00	05	00	00
	ADD HL,DE	08	01	0	00	05	00	05
NOADD	SLA E	08	01	0	00	0A	00	05
	RL D	08	01	0	00	0A	00	05
	DEC B	07	01	0	00	0A	00	05
	JP NZ,010F	07	01	0	00	0A	00	05

Figure 3.23. — Une passe dans la boucle

L'instruction SRL effectue un décalage logique vers la droite. Le bit le plus à droite de MPR tombe dans l'indicateur C. La figure 3.23 montre que le contenu de MPR, après le décalage, est « 00000001 », et que l'indicateur C vaut maintenant « 1 ». Les autres registres n'ont pas été affectés par cette opération. Continuez maintenant de remplir ce tableau par vous-même.

Un second passage dans la boucle est présenté à la figure 3.41.

La liste complète des contenus des registres du Z80 et des indicateurs est présentée à la figure 3.39, à la fin de ce chapitre, pour la multiplication complète. Un listing hexadécimal est présenté à la figure 3.40.

Les choix en programmation

Il existe bien d'autres manières d'écrire le programme précédent. En règle générale, tout programmeur peut aisément trouver les moyens de modifier, et souvent d'améliorer, un programme. Nous avons, par exemple, décalé le multiplicande à gauche, avant l'addition. Il aurait été mathématiquement équivalent de décaler à droite le résultat, toujours avant l'addition.

C'est, en fait, un exercice intéressant !

Exercice 3.19 : *Ecrire un programme de multiplication 8 × 8 en utilisant le même algorithme, mais en décalant le résultat d'une position vers la droite au lieu de décaler le multiplicande vers la gauche. Comparez-le au programme précédent, et déterminez si cette nouvelle approche est plus rapide, ou plus lente, que la précédente. Les vitesses d'exécution des instructions du Z80 seront données au prochain chapitre.*

Programme de multiplication amélioré

Le programme développé est une traduction directe de l'algorithme. Cependant, *une programmation efficace exige de porter une attention soutenue à tous les détails.* La longueur du programme peut souvent être réduite, et sa vitesse améliorée. Nous allons étudier différentes manières d'améliorer notre programme de base.

1ère étape

Première amélioration possible : une meilleure utilisation du jeu d'instructions. Les avant-dernière et antépénultième instructions peuvent être remplacées par une instruction unique :

DJNZ MULT

Il s'agit d'une instruction de saut automatique, propre au Z80, qui décrémente le registre B et se branche à l'adresse spécifiée, si B n'a pas

atteint la valeur 0. Pour être tout à fait correct, il faut préciser qu'elle n'est pas strictement équivalente à la paire :

```
DEC B
JP  NZ, MULT
```

puisqu'elle indique un *déplacement,* et qu'un tel saut ne peut avoir lieu que dans l'intervalle − 128 à + 128. Toutefois, notre saut n'excède pas ici quelques adresses, et l'amélioration proposée est donc parfaitement légitime. Pour le programme qui en résulte, voir figure 3.24.

```
MUL88B  LD    DE, (MPDAD)
        LD    D, 0
        LD    BC, (MPRAD)
        LD    B, 8
        LD    D, 0
        LD    HL, 0
MULT    SRL   C
        JR    NC, NOADD
        ADD   HL, DE
NOADD   SLA   E
        RL    D
        DJNZ  MULT
        LD    (RESAD), HL
        RET
```

Figure 3.24. — Etape 1 de l'amélioration de la multiplication

2e étape

Pour procéder à une nouvelle amélioration du programme, remarquons que trois opérations différentes de décalage ont lieu dans le programme initial, présenté à la figure 3.13. Le multiplicateur est décalé à droite, et le multiplicande à gauche, en deux temps : d'abord en décalant E à gauche, puis en effectuant une rotation à gauche au registre D. Tout cela prend du temps. Un « truc » bien connu en programmation consiste à remarquer que chaque fois que le multiplicateur est décalé d'un bit vers la droite, un autre bit devient disponible à gauche du registre. Simultanément, on peut observer que le premier produit partiel utilise, au maximum, 9 bits. Si, au début du programme, un seul registre avait été affecté au résultat, nous aurions pu nous servir du bit libéré dans le multiplicateur pour ranger le neuvième bit du résultat.

Après le décalage suivant de MPR, la taille du produit partiel est encore augmentée d'un bit. Au fur et à mesure des décalages de MPR les bits libérés pourront être utilisés au rangement des produits partiels successifs. L'amélioration consistera donc à mettre MPR et RES dans la même paire

de registres. Idéalement, ils devraient être décalés tous deux en une seule opération. Malheureusement, le Z80 ne peut décaler que 8 bits à la fois. Comme beaucoup d'autres microprocesseurs, il ne dispose pas d'instructions permettant de décaler 16 bits simultanément.

Cependant, un autre « truc » peut être utilisé. Le Z80 (comme le 8080) possède des instructions sur 16 bits, que nous avons d'ailleurs déjà utilisées. Si MRP et RES sont tous deux dans la paire HL, nous pourrons faire intervenir :

ADD HL, HL

qui additionne HL à lui-même. Additionner un nombre à lui-même, c'est le doubler. Doubler un nombre, dans le système binaire, équivaut à le décaler à gauche d'une position. Nous avons ainsi trouvé le moyen de décaler, d'un coup, 16 bits d'une position. Malheureusement, ce décalage a lieu vers la gauche, alors que nous l'attendions vers la droite. Le problème n'est nullement insoluble.

Conceptuellement, le MPR peut être indifféremment décalé vers la droite ou vers la gauche. Nous avons utilisé un algorithme à décalage à droite, parce que c'est lui qui intervient dans l'addition ordinaire. Cependant, ce choix n'est pas une obligation. L'addition est commutative, et l'ordre des opérations peut donc être inversé : décaler MPR vers la gauche est parfaitement possible.

Pour bénéficier de tous les avantages de ce décalage simulé sur 16 bits, nous allons décaler MPR vers la gauche. Le MPR sera ainsi dans H, et le résultat dans L.

La figure 3.25 montre la nouvelle organisation des registres.

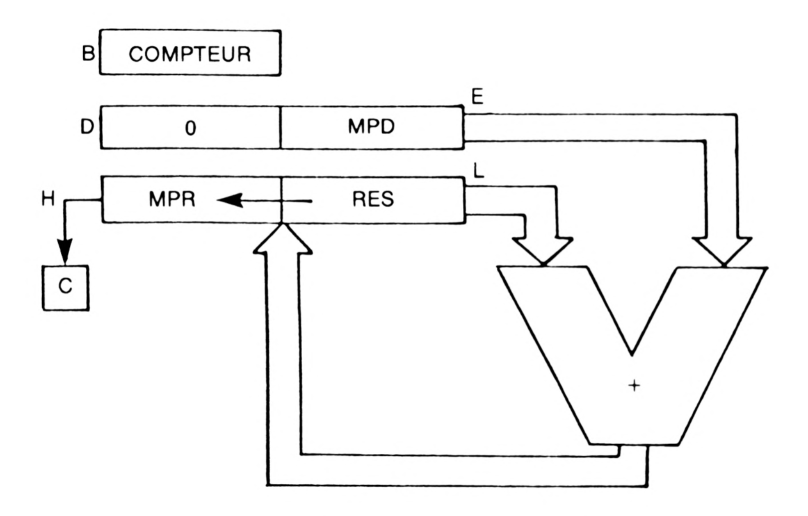


Figure 3.25. — Les registres pour la multiplication améliorée

Le reste du programme est pratiquement identique au précédent.

Le nouveau programme est le suivant :

```
MUL88C  LD    HL, (MPRAD - 1)
        LD    L, 0
        LD    DE, (MPDAD)
        LD    D, 0
        LD    B, 8        COMPTEUR
MULT    ADD   HL, HL         DECALAGE A GAUCHE
        JR    NC, NOADD
        ADD   HL, DE
NOADD   DJNZ  MULT
        LD    (RESAD), HL
        RET
```

Figure 3.26. — Multiplication améliorée, 2e étape

Comparons ce programme au précédent : il apparaît que la longueur de la boucle de multiplication (le nombre d'instructions entre MULT et le saut) a diminué. Ce programme comprend moins d'instructions, et son exécution devrait donc être en principe, plus rapide. Nous venons de voir que le choix des registres contenant l'information est important.

Une conception directe suscitera généralement un programme approprié, mais pas *optimisé*. Il est donc important de comprendre la manière d'utiliser les registres et les instructions disponibles du mieux possible. Ces exemples sont une illustration d'une approche rationnelle du choix des registres et des instructions, mue par un souci d'efficacité.

Exercice 3.20 : *Calculer la vitesse de la multiplication dans le dernier programme. Nous supposerons que le branchement a lieu dans 50 % des cas. Vous trouverez le nombre de cycles demandés par chaque instruction dans l'index. La fréquence de l'horloge sera fixée à 2 MHz (un cycle = 0.5 µs).*

Exercice 3.21 : *Nous avons eu recours à la paire DE pour le multiplicande. Comment le programme ci-dessus serait-il modifié si nous utilisions, à la place, la paire BC ? (Conseil : cette substitution provoquerait une modification, à la fin).*

Exercice 3.22 : *Pourquoi faut-il se préoccuper de mettre à 0 le registre D après avoir chargé MPD dans E ?*

Pour finir, occupons-nous d'un problème qui paraîtra peut-être irritant au programmeur peu familier du Z80. Le lecteur aura remarqué que pour charger, depuis la mémoire, MPD dans E, il est nécessaire de charger simultanément D et E, à partir de deux adresses consécutives. Pourquoi ? Parce qu'il n'y a pas moyen d'aller chercher un seul octet en mémoire pour le ranger dans E à moins que l'adresse mémoire ne soit contenue dans HL. C'est un héritage du vieux 8008, qui ne possédait pas l'adressage direct. Cette caractéristique se retrouve, sous forme améliorée dans le 8080, et,

pourvue de nouvelles améliorations, dans le Z80, où il est possible d'aller chercher directement 16 bits à une adresse donnée en mémoire (mais pas 8 bits).

Ce petit mystère résolu, passons à une multiplication plus complexe.

Une multiplication 16 × 16

Pour mettre à l'épreuve les techniques nouvellement acquises, nous tenterons de multiplier deux nombres de 16 bits, en supposant cependant que le résultat tient sur 16 bits, de telle sorte qu'il puisse être contenu dans une paire de registres.

Le résultat, comme dans notre premier exemple de multiplication, sera contenu dans la paire HL (voir figure 3.27). Le multiplicande MPD sera dans DE.

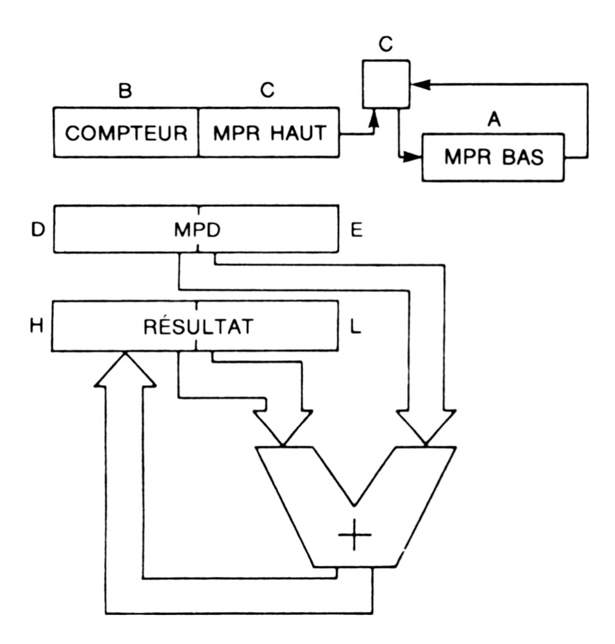


Figure 3.27. — Multiplication 16 × 16 — Les registres

Il serait tentant de mettre le multiplicateur dans BC. Cependant, pour tirer parti de l'instruction DJNZ, le registre B doit servir de compteur. Par

conséquent, le registre C contiendra une moitié du multiplicateur, et A l'autre moitié (voir figure 3.27). Voici le programme de multiplication :

```
MULT16  LD    A, (MPRAD + 1)  MPR, PARTIE HAUTE
        LD    C, A
        LD    A, (MPRAD)      MPR, PARTIE BASSE
        LD    B, 16           COMPTEUR
        LD    DE, (MPDAD)     MPD
        LD    HL, 0
MULT    SRL   C               DECALAGE A DROITE DE
                              MPR HAUT
        RRA                   ROTATION A DROITE DE
                              MPR BAS
        JR    NC, NOADD       TESTER L'INDICATEUR C
        ADD   HL, DE
NOADD   EX    DE, HL
        ADD   HL, HL          DECALAGE A GAUCHE DE
                              MPD
        EX    DE, HL
        DJNZ  MULT
        RET
```

Figure 3.28. — Programme de multiplication 16 × 16

Ce programme est analogue aux précédents. Les six premières instructions (de MULT16 à MULT) initialisent les registres avec les valeurs appropriées. Une petite complication est introduite par l'obligation de charger les deux moitiés de MPR séparément. Nous supposerons que MPRAD pointe en mémoire sur la partie basse de MPR, et que la partie haute se trouve à l'adresse suivante ; nous aurions pu postuler la convention contraire. Lorsque la partie haute de MPR est chargée dans A, elle doit être transférée dans C :

```
        LD    A, (MPRAD + 1)
        LD    C, A
```

Finalement, la partie basse de MPR est lue directement dans A :

```
        LD    A, (MPRAD)
```

Les autres registres sont initialisés comme à l'accoutumée :

```
        LD    B, 16
        LD    DE, (MPDAD)
        LD    HL, 0
```

Un décalage d'une largeur de 16 bits devra être opéré sur le multiplicateur. Chaque décalage d'une position a lieu en deux temps :

```
        MULT  SRL  C
              RRA
```

Ensuite, le bit le plus à droite du MPR est déposé dans l'indicateur C, où il peut être testé :

```
              JR     NC, NOADD
```

Comme d'habitude, le multiplicande n'est pas ajouté au résultat si l'indicateur C vaut « 0 ». Il le sera si C vaut « 1 ».

```
              ADD    HL, DE
```

Décalons maintenant le MPD d'une position vers la gauche.

Le Z80 ne possède pas d'instruction capable de décaler simultanément vers la gauche le contenu de la paire DE d'une position, et pas davantage d'instructions permettant d'additionner DE à lui-même. Nous transférerons alors le contenu de DE dans HL, avant de doubler ce dernier et de transférer son contenu dans DE, au moyen des trois instructions suivantes :

```
NOADD   EX     DE, HL
        ADD    HL, HL
        EX     DE, HL
```

L'instruction EX (de l'anglais EXchange : échange) est chargée d'échanger les contenus de HL et de DE. Après exécution de EX DE, HL, le contenu de HL a été transféré dans DE, et inversement. La valeur du résultat n'est pas, pour autant, perdue. Elle se trouve provisoirement sauvegardée dans DE.

Le compteur B est décrémenté. Un saut se produit vers le début de la boucle aussi longtemps que B n'a pas atteint la valeur « 0 ».

```
        DJNZ   MULT
```

Bien sûr, d'autres répartitions des registres peuvent, éventuellement, donner lieu à des programmes plus courts.

Exercice 3.23 : *Mettre le multiplicateur dans les registres B et C, et le compteur dans A. Ecrivez le programme de multiplication correspondant, et discutez des avantages et des inconvénients de cette allocation de registres.*

Exercice 3.24 : *Reportez-vous au programme de multiplication 16 bits de la figure 3.28, et proposez un moyen de décaler le MPD contenu dans les registres D et E, sans le transférer dans la paire HL.*

Exercice 3.25 : *Ecrivez un programme de multiplication 16 bits par 16 qui précise si le résultat tient sur plus de 16 bits. Il s'agit d'une simple amélioration de notre programme de base.*

Exercice 3.26 : *Ecrire un programme de multiplication 16 par 16 dont le résultat tienne sur 32 bits. La méthode suggérée d'allocation des registres apparaît à la figure 3.29. Rappelez-vous que le premier résultat partiel (après la première addition) tient sur 16 bits, et qu'à chaque passage dans la boucle le multiplicateur libère un bit.*

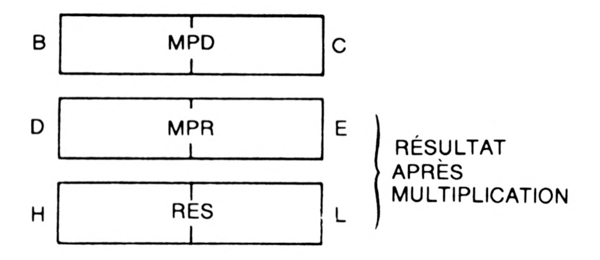


Figure 3.29. — Multiplication 16 × 16 avec résultat 32 bits

Nous allons maintenant étudier la dernière opération classique, la division.

Division binaire

L'algorithme de la division binaire est analogue à celui de la multiplication. Le diviseur est successivement soustrait des bits de poids fort du dividende. Après chaque opération, le résultat est utilisé à la place du dividende initial, et la valeur du quotient est, simultanément, augmentée de 1. Le résultat de la soustraction devient, en définitive, négatif. Il faut alors restaurer le résultat partiel, en ajoutant le diviseur. Naturellement, le quotient devra alors être décrémenté de 1. Quotient et dividende sont décalés d'une position vers la gauche, et l'algorithme est répété. L'ordinogramme figure à la figure 3.30.

La méthode que nous venons de décrire est dite de division avec *restauration.* Il existe une variante plus rapide : la division *sans restauration.*

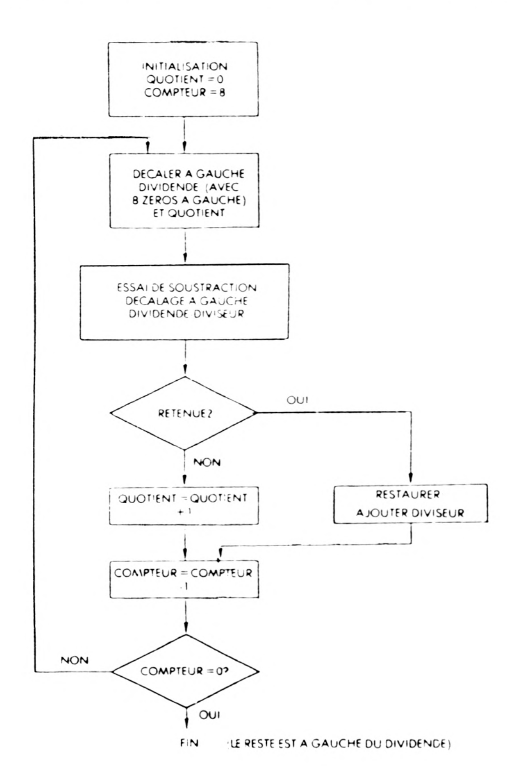


Figure 3.30. — Ordinogramme division binaire 8 bits

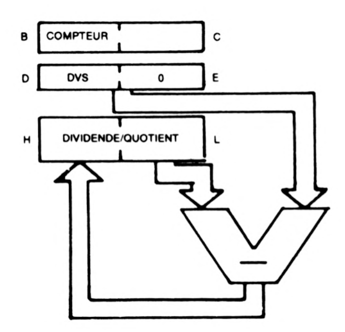


Figure 3.31. — Division 16 × 8 — Les registres

Division 16 × 8

A titre d'exemple, examinons la division 16 par 8, qui produira un quotient sur 8 bits, et un reste sur 8 bits également. L'utilisation des registres est présentée à la figure 3.31. Voici le programme :

```
DV168   LD    A, (DVSAD)     DIVISEUR
        LD    D, A           DANS D
        LD    E, 0
        LD    HL, (DVDAD)    DIVIDENDE SUR 16 BITS
        LD    B, 8
DIV     XOR   A              INDICATEUR C A ZERO
        SBC   HL, DE         DIVIDENDE — DIVISEUR
        INC   HL             QUOTIENT = QUOTIENT + 1
        JP    P, NOADD       RESTE POSITIF ?
        ADD   HL, DE         RESTAURER SI NECESSAIRE
        DEC   HL
NOADD   ADD   HL, HL         DECALER DIVIDENDE
        DJNZ  DIV            BOUCLER JUSQU'A B = 0
        RET
```

Figure 3.32. — Programme de division 16 × 8

Les cinq premières instructions du programme chargent respectivement le diviseur et le dividende dans D et HL. Elles initialisent aussi le compteur (registre B) à 8. A noter la fréquente utilisation de B comme compteur, lorsque l'instruction DJNZ peut être mise en œuvre.

```
DV168   LD    A, (DVSAD)
        LD    D, A
        LD    E, 0
        LD    HL, (DVDAD)
        LD    B, 8
```

Soustrayons le diviseur du dividende. Comme nous sommes contraints de recourir à l'instruction SBC (il n'y a pas de soustraction 16 bits qui ne tienne compte de l'indicateur C), il convient, au préalable, de mettre l'indicateur C à « 0 ». L'opération peut être mise en œuvre de différentes et fort nombreuses façons. Par exemple, à l'aide des instructions :

```
XOR   A
AND   A
OR    A
```

Nous avons, ici, utilisé XOR A :

```
DIV     XOR A
```

La soustraction suivante est ensuite effectuée :

```
SBC     HL, DE
```

Nous postulerons, a priori, qu'elle réussira, autrement dit que son résultat sera positif. C'est ce qu'on appelle « l'étape d'essai » (voir l'ordinogramme de la figure 3.30). Le quotient est ensuite incrémenté de 1. Si, au contraire, la soustraction échoue (produit un reste négatif), il nous faudra plus loin décrémenter à nouveau le quotient.

```
        INC     HL
```

Testons le résultat de la soustraction

```
        JP      P, NOADD
```

Si le reste est positif ou nul, la soustraction a réussi, et il n'est pas nécessaire de restaurer. Le programme saute à l'adresse NOADD. Dans le cas contraire l'ancienne valeur du dividende doit être restaurée, en lui ajoutant le diviseur, et le quotient décrémenté, au moyen des deux instructions :

```
        ADD     HL, DE
        DEC     HL
```

Le dividende restant est décalé d'une position vers la gauche, en prévision du prochain essai de soustraction. Le compteur B est décrémenté, et on teste s'il atteint la valeur « 0 » ; tant que tel n'est pas le cas, la boucle continue :

```
NOADD   ADD     HL, HL
        DJNZ    DIV
        RET
```

Exercice 3.27 : *Vérifier, à la main, le fonctionnement de ce programme de division, en remplissant le tableau de la figure 3.33, de la même manière que pour la multiplication, dans l'exercice 3.18.*

Remarquez que le registre D ne figure pas sur le tableau : son contenu n'est jamais modifié.

ETIQUETTE	INSTRUCTION	B	H	L

Figure 3.33. — Tableau pour le programme de division

Division 8 bits

Le programme suivant utilise une méthode avec restauration, et laisse un quotient complémenté dans A. Il effectue une division non-signée, 8 bits par 8 bits.

E est le dividende
C est le diviseur
A est le quotient
B est le reste

```
DIV88      XOR   A            ZERO DANS L'ACCUMU-
                              LATEUR
           LD    B, 8         COMPTEUR
BOUCLE88   RL    E            ENTRE L'INDICATEUR C
                              DANS LE DIVIDENDE
           RLA                LA PREMIERE FOIS, INDI-
                              CATEUR C A ZERO
           SUB   C            ESSAI DE SOUSTRACTION
                              DU DIVISEUR
           JR    NC, $ + 3    SAUT SI SOUSTRACTION
                              OK
           ADD   A, C         RESTAURE A, MET INDI-
                              CATEUR C A 1
           DJNZ  BOUCLE88
           LD    B, A         MET LE RESTE DANS B
           LD    A, E         QUOTIENT
           RLA                ENTRE LE DERNIER BIT
                              DU RESULTAT
           CPL                COMPLEMENTE A
           RET
```

Remarque : à la sixième instruction, le symbole $ représente la valeur du compteur ordinal.

Division sans restauration

Le programme suivant effectue une division entière, 16 bits par 15 bits, au moyen d'une méthode sans restauration. IX pointe sur le dividende, et IY sur le diviseur (qui ne doit pas être nul). Les adresses sont laissées dans IX et IY (figure 3.34).

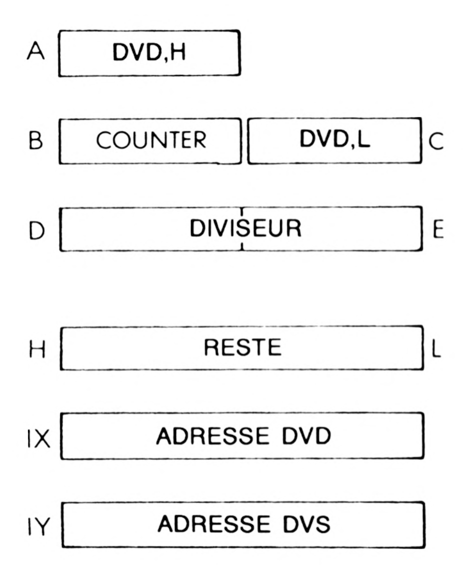


Figure 3.34. — Division sans restauration. Les registres

Le registre B est utilisé comme compteur et initialise à 16.
A et C contiennent le dividende.
D et E contiennent le diviseur.
H et L contiennent le résultat.
Le dividende sur 16 bits est décalé, à gauche, par les instructions :

```
RL	C
RLA
```

Le reste est décalé à gauche par :

```
	ADC	HL, HL
```

Le quotient final est laissé dans B, C, et le reste dans HL. Voici le programme :

```
DIV16     LC    B, (IX + 1)
          LD    C, (IX)
          LD    D, (IY + 1)
          LD    E, (IY)
          LD    A, D          DIVISEUR NUL SI
          OR    E             (A) OR (E) = 0
          JR    Z, ERREUR     DIVISEUR NUL INTERDIT
          LD    A, B          DIVIDENDE, PARTIE
                              HAUTE
          LD    HL, 0         INITIALISE LE RESULTAT
          LD    B, 16         COMPTEUR
ESS-SOUST RL    C             ROTATION A GAUCHE DU
                              DIVIDENDE
          RLA
          ADC   HL, HL        DECALAGE A GAUCHE —
                              NE POSITIONNE JAMAIS
                              L'INDICATEUR C
          SBC   HL, DE        SOUSTRACTION DU DIVI-
                              SEUR
NUL       CCF                 BIT RESULTAT
          JR    NC, NGV       ACCUMULATEUR NEGA-
                              TIF ?
PTV       DJNZ  ESS-SOUST     COMPTEUR A ZERO ?
          JP    FINI
RESTAURE  RL    C             ROTATION DIVIDENDE A
                              GAUCHE
          RLA                 COMME CI-DESSUS
          ADC   HL, HL
          AND   A
          ADC   HL, DE        RESTAURER PAR AJOUT
                              DU DIVISEUR
          JR    C, PTV        SAUT SI RESULTAT
                              POSITIF
          JR    Z, NUL        RESULTAT NUL
NGV       DJNZ  RESTAURE      COMPTEUR A ZERO ?
FINI      RL    C             ENTRE LE DERNIER BIT
                              DU RESULTAT
          RLA                 POUR OBTENIR LE RESTE
          ADD   HL, DE        CORRECT
          LD    B, A          QUOTIENT DANS BC
          RET
```

Exercice 3.28 : *Comparer le programme précédent à celui qui suit. Ce dernier utilise, quant à lui, une technique avec restauration.*

Dividende dans AC
Diviseur dans DE
Quotient dans AC
Reste dans HL

```
DIV16   LD    HL, 0       INITIALISE HL A 0
        LD    B, 16       INITIALISE LE COMPTEUR
BOUCLE  RL    C           ROTATION DIVIDENDE/
                          RESULTAT
        RLA
        ADC   HL, HL      DECALAGE A GAUCHE
        SBC   HL, DE      ESSAI DE SOUSTRACTION
                          DU DIVISEUR
        JR    NC, $ + 3   SAUT SI SOUSTRACTION OK
        ADD   HL, DE      SINON RESTAURER DIVI-
                          DENDE
        CCF               CALCULER LE BIT
                          RESULTAT
        DJNZ  BOUCLE      BOUCLER SI PAS FINI
        RL    C           ENTRER LE DERNIER BIT
        RLA
        RET
```

Remarque : Le SYMBOLE $ désigne « l'adresse courante » (i.e. la valeur courante du compteur ordinal).

OPÉRATIONS LOGIQUES

Autre classe d'instructions pouvant être exécutées par l'ALU : les *instructions logiques.* Elle comprend : AND, OR et XOR (ou exclusif). Les instructions de décalage et de rotation, que nous avons déjà utilisées, et l'instruction de comparaison, appelée CP dans le Z80, peuvent également y être incluses. L'utilisation individuelle de AND, OR et XOR sera décrite au chapitre 4, consacré aux instructions.

Ecrivons un programme qui testera la valeur du contenu d'un emplacement mémoire donné : LOC. Cette valeur pourra être « 0 », « 1 », ou autre.

Le programme introduit l'instruction de comparaison, et accomplit une série de tests logiques. Suivant le résultat de la comparaison, tel segment, ou tel autre, sera exécuté.

Voici le programme :

```
                LD      A, (LOC)    LIT LE CONTENU DE LOC
                CP      00H         COMPARE AVEC 0
                JP      Z, ZERO     SAUT SI EGAL A 0
                CP      01H         COMPARE A 1
                JP      Z, UN       SAUT SI EGAL A 1
AUTRECHOSE      .....
                .....
                .....
ZERO            .....
                .....
                .....
UN              .....
```

La première instruction, LD A, (LOC), lit le contenu de la case mémoire LOC, et le charge dans l'accumulateur. C'est la valeur à tester, que nous comparons à la valeur 0 par l'instruction :

```
CP      00H
```

Cette instruction compare le contenu de l'accumulateur à la valeur hexadécimale « 00 », autrement dit à la valeur binaire « 0000 0000 ». Elle met l'indicateur Z à la valeur « 1 », si la comparaison réussit. Cet indicateur peut être testé par l'instruction suivante :

```
JP      Z, ZERO
```

Cette instruction de saut, dite conditionnelle, teste la valeur de l'indicateur Z. Si la comparaison réussit, le bit Z est mis à 1, et le branchement a lieu ; le programme continue alors à l'adresse ZERO. Si elle échoue, le branchement ne s'effectue pas, et le programme continue en séquence :

```
CP      01H
```

De la même manière, l'instruction de saut suivante provoque un branchement à l'adresse UN, si la comparaison réussit. Si les deux comparaisons échouent, l'instruction située à l'adresse AUTRECHOSE est exécutée.

```
                JP      Z, UN
AUTRECHOSE      .....
```

Ce programme a été introduit pour montrer l'intérêt de la succession d'une instruction de comparaison et d'une instruction de saut. Cette combinaison sera utilisée dans un grand nombre de programmes à venir.

Exercice 3.29 : *Référez-vous à la définition de l'instruction LD A, (LOC), au chapitre suivant. Regardez l'effet éventuel de cette instruction sur le registre d'indicateurs (F). La seconde instruction du programme, CP 00H est-elle nécessaire ?*

Exercice 3.30 : *Ecrire un programme qui lise le contenu de l'adresse mémoire « 24 », et se branche à l'adresse appelée ETOILE si le contenu de la case mémoire « 24 » est « * ». Nous supposerons que la représentation binaire du caractère « * » est « 0010 1010 ».*

CONCLUSION SUR LES INSTRUCTIONS

Nous avons maintenant étudié, en les mettant en œuvre, la plupart des instructions importantes du Z80. Des valeurs ont été transférées entre la mémoire et les registres, et des opérations arithmétiques et logiques effectuées sur des données. Ces données ont été testées et, suivant le résultat, diverses parties des programmes ont été exécutées. Nous avons, en particulier, utilisé des instructions « automatiques » du Z80, telles que DJNZ, pour raccourcir les programmes. D'autres instructions automatiques, telles que LDDR, CPIR, INIR, seront introduites au cours de ce livre.

Plein usage a donc été fait des caractéristiques particulières du Z80. Le lecteur aura soin de ne pas utiliser les programmes présentés sur un 8080 : ils ont été optimisés pour le Z80. Après la boucle, une autre structure de programmation importante va maintenant être introduite : les sous-programmes.

LES SOUS-PROGRAMMES

Conceptuellement, un sous-programme est tout simplement un bloc d'instructions auquel le programmeur a donné un nom. Il se termine par une instruction spéciale, appelée *retour*. Nous démontrerons d'abord l'intérêt de l'utilisation d'un sous-programme dans un programme, avant d'en étudier le fonctionnement.

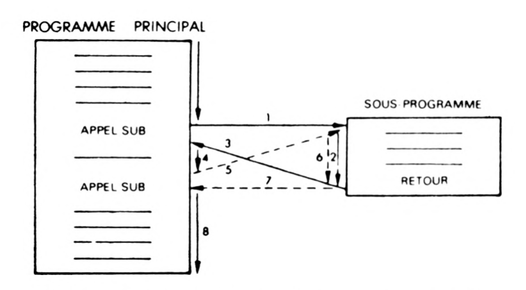


Figure 3.35. — Appels de sous-programmes

L'utilisation d'un sous-programme est présentée à la figure 3.35. Le programme principal apparaît à gauche de l'illustration. Le sous-programme est représenté, de manière symbolique, sur la droite.

Examinons le mécanisme du sous-programme. Les lignes du programme principal sont exécutées normalement, jusqu'à la rencontre d'une nouvelle instruction : « CALL SUB ». Cette instruction spéciale est un *appel de sous-programme :* elle produit un branchement vers le sous-programme, à l'adresse SUB. Cela signifie que l'instruction exécutée immédiatement après « CALL SUB » est la première instruction du sous-programme, comme le montre la flèche 1, sur la figure.

Les instructions du sous-programme s'exécutent ensuite, exactement de la même manière que celles d'un programme quelconque. Nous supposerons ici que notre sous-programme ne comporte pas d'instruction CALL. La dernière instruction du sous-programme est RETURN, une instruction spéciale qui provoque un retour au programme principal, comme le montre la flèche 3, sur la figure. L'exécution du programme se poursuit, ensuite, normalement (flèche 4).

Dans le corps du programme principal, un second « CALL SUB » apparaît. D'où un nouveau transfert (flèche 5) vers l'adresse SUB.

De nouveau, les instructions du sous-programme vont être exécutées.

Lors de la rencontre de l'instruction RETURN un retour au programme principal a lieu, immédiatement derrière le CALL SUB qui est à l'origine de l'exécution du sous-programme (flèche 7). L'exécution du programme principal reprend alors normalement (flèche 8).

L'effet de ces deux instructions spéciales, CALL et RETURN devrait maintenant être clair. Quel est l'intérêt de ce mécanisme de sous-programmes ?

En premier lieu, un sous-programme peut être appelé à partir d'un nombre quelconque de points du programme principal, et être utilisé plusieurs fois *sans réécriture.* Premier avantage : cette méthode économise de la mémoire, puisque la même séquence d'instructions peut être exécutée plusieurs fois sans être réécrite. Second avantage : le programmeur conçoit un sous-programme spécifique, une fois pour toutes, de façon à l'utiliser quand le besoin s'en fait sentir. C'est une simplification importante de la programmation.

Exercice 3.31 : *Quel est le principal désavantage d'un sous-programme ?*

Réponse : Le désavantage d'un sous-programme apparaît clairement lorsqu'on examine la séquence des instructions exécutées par le programme principal et par le sous-programme. La présence d'un sous-programme *ralentit l'exécution,* puisque chaque exécution des instructions du sous-programme nécessite la présence de deux instructions supplémentaires : CALL et RETURN.

Mécanismes des sous-programmes

Nous nous intéresserons, maintenant, à la manière dont les deux instructions, CALL SUB et RET, sont exécutées dans le microprocesseur.

L'effet de CALL SUB est de provoquer la recherche de l'instruction suivante, à une nouvelle adresse.

On se souvient (voir chapitre 1) que l'adresse de la prochaine instruction à exécuter est contenue dans le compteur ordinal (PC). L'instruction CALL SUB substitue donc une nouvelle valeur au registre PC. Elle charge l'adresse du début du sous-programme (SUB) dans le compteur ordinal. *Est-ce vraiment suffisant ?*

Pour répondre à cette question, intéressons-nous au rôle de l'instruction RETURN. Cette instruction doit provoquer, comme son nom l'indique, un retour vers l'instruction faisant suite à CALL SUB. A condition que son adresse ait été sauvegardée quelque part. Il se trouve qu'elle correspond justement à la valeur du compteur ordinal, après que la totalité de l'instruction CALL SUB ait été lue par l'unité de contrôle du microprocesseur. L'effet de l'instruction CALL est donc double : d'abord sauvegarder, quelque part, la valeur du compteur ordinal, puis opérer le branchement à l'adresse indiquée.

Le problème est alors le suivant : où sauvegarder cette adresse de retour ? Il faut la mettre dans un endroit où elle n'ait aucune chance d'être effacée.

Examinons le cas, présenté sur la figure 3.36, où le sous-programme 1 contient un appel au sous-programme 2. La validité de notre mécanisme ne doit pas être mise en cause. Naturellement, il pourrait y avoir encore plus de sous-programmes, mettons, « N » sous-programmes imbriqués. A chaque CALL, le mécanisme doit sauvegarder une adresse de retour, ce qui implique que nous disposions de 2N cases mémoire. De plus, il importe de pouvoir revenir, d'abord de SUB2, puis de SUB1. En d'autres termes, nous avons besoin d'une structure capable de préserver l'ordre chronologique dans lequel les adresses ont été préservées.

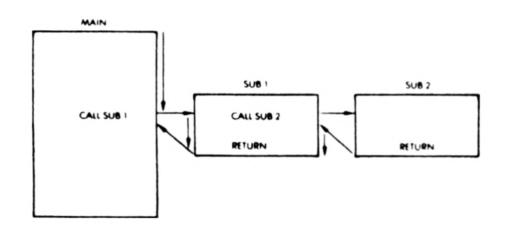


Figure 3.36. — Appels imbriqués

Nous connaissons déjà cette structure. Il s'agit de la *pile*. La figure 3.38 montre ses contenus successifs au cours des différents appels de sous-programmes. Examinons d'abord le programme principal. Le premier appel, CALL SUB1, se situe à l'adresse 100. Nous supposerons que, dans ce microprocesseur, l'appel du sous-programme tient sur trois octets (RST est une exception). L'instruction suivante se trouve, non pas à l'adresse « 101 », mais à l'adresse « 103 ». CALL utilise les adresses « 100 », « 101 » et « 102 ». Parce que l'unité de contrôle du Z80 « sait » qu'il s'agit d'une

instruction sur trois octets, la valeur du compteur ordinal, après décodage complet de l'instruction, sera « 103 ». L'effet du CALL SUB1 va être de charger la valeur « 280 » dans le compteur ordinal ; « 280 » étant l'adresse du début du sous-programme SUB1.

Nous sommes maintenant prêts à présenter l'effet de l'instruction RETURN, et le fonctionnement correct de notre mécanisme de pile. L'exécution continue, dans SUB2, jusqu'à la rencontre de l'instruction RETURN. L'effet de cette dernière est simplement de dépiler le sommet de la pile dans le compteur ordinal. En d'autres termes, le compteur ordinal va retrouver la valeur qui était la sienne juste avant l'entrée dans le sous-programme. Le sommet de la pile se trouve être, dans notre exemple, « 303 ». La figure 3.38 montre qu'au temps 3, la valeur « 303 » a été retirée de la pile et remise dans le compteur ordinal. L'exécution reprend ainsi à l'adresse « 303 ». Au temps 4 : rencontre du RETURN de SUB1. La valeur au sommet de la pile est « 103 ». Elle est dépilée et mise dans le compteur ordinal. L'exécution reprend à l'adresse « 103 » du programme principal. C'est, en fait, l'effet recherché. La figure 3.38 montre qu'au temps 4, la pile est de nouveau vide. Le mécanisme fonctionne.

Le mécanisme d'appel de sous-programme ne peut fonctionner que jusqu'à la taille maximum de la pile. C'est pourquoi les premiers microprocesseurs, qui ne disposaient que de 4 ou 8 registres de pile, étaient limités à 4 ou 5 niveaux d'appels de sous-programmes.

Remarque : sur les figures 3.36 et 3.37, les sous-programmes ont été représentés à la droite du programme principal, pour une raison de clarté de présentation. En réalité, l'utilisateur les écrira comme des instructions tout à fait normales. Sur le listing d'un programme complet, les sous-programmes peuvent se trouver au début, au milieu ou à la fin du texte.

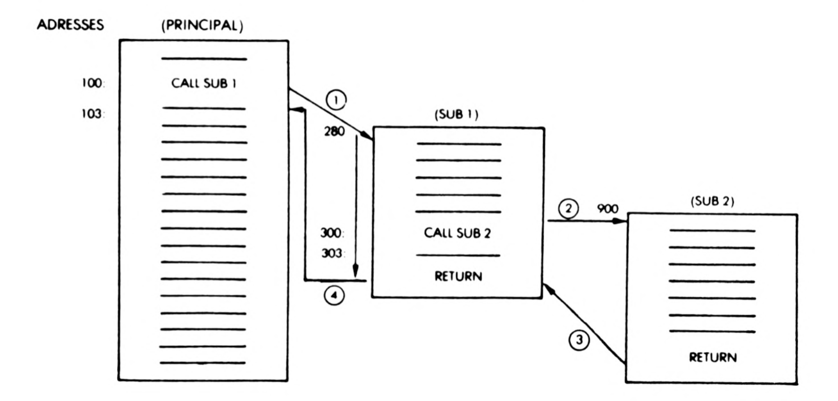


Figure 3.37. — Appels de sous-programmes

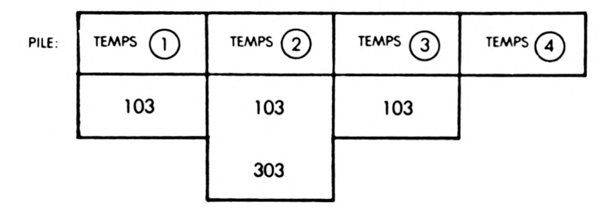


Figure 3.38. — La pile en fonction du temps

Les sous-programmes dans le Z80

Les concepts de base relatifs aux sous-programmes ont maintenant été présentés. Nous avons vu que la pile est nécessaire à la mise en œuvre du mécanisme. Le Z80 possède un registre pointeur de pile de 16 bits. La pile peut se trouver à n'importe quel endroit de la mémoire, et atteindre une taille de 64 K octets (1 K = 1024), en supposant qu'un si grand nombre d'octets soit disponible pour cet usage. En pratique, l'adresse de départ de la pile, et sa taille maximum, seront définies par le programmeur, avant l'écriture du programme. Une zone mémoire sera réservée à la pile.

L'instruction d'appel de sous-programme, dans le cas du Z80, s'appelle CALL. Elle comporte deux versions. L'appel direct ou inconditionnel, tel que CALL ADRESSE, est la version que nous venons d'utiliser. Mais le Z80 possède de surcroît une instruction d'appel conditionnel, qui n'entrera en jeu que si une certaine condition est vérifiée. Par exemple :

CALL NZ, SUB1 ne provoquera l'appel au sous-programme 1 que si le résultat de l'opération précédente est différent de zéro. C'est une caractéristique puissante, car en pratique, beaucoup d'appels de sous-programmes sont conditionnels.

CALL CC, NN ne sera exécutée que si la condition spécifiée par « CC » est vérifiée. CC est un ensemble de trois bits (bits 4, 5 et 6 du code opération) capables de spécifier l'une des huit conditions. Ces conditions correspondent, respectivement, à l'état « 1 » ou « 0 » des quatre indicateurs « Z », « C », « P/V » et « S ».

De la même manière, il existe deux types d'instructions de retour : RET et RET CC.

RET est l'instruction de base. Elle occupe un octet, et provoque le dépilage, dans le compteur ordinal, des deux octets formant le sommet de la pile. C'est une instruction inconditionnelle.

RET CC provoque le même effet, à ceci près qu'elle n'est exécutée que si la condition indiquée par CC est vérifiée. Les bits de condition sont les mêmes que ceux utilisés pour l'instruction CALL.

On dénombre aussi deux types de retours spécialisés, chargés de terminer les routines d'interruption RETI, RETN. Ils sont décrits dans les chapitres consacrés aux instructions et aux interruptions.

Enfin, une instruction plus spécialisée, analogue à l'appel d'un sous-programme, permet de se brancher uniquement à l'une des huit adresses situées en page 0. Il s'agit de l'instruction RST P, sur un octet, qui sauvegarde automatiquement le contenu du compteur ordinal sur la pile, et provoque un branchement à l'adresse [sur trois bits] spécifiée par le champ P. Ce dernier correspond à la valeur des bits 3, 4 et 5 de l'instruction, multipliée par 8.

Autrement dit, si les bits 3, 4 et 5 valent « 000 », le saut se produira vers l'adresse 00H. S'ils valent « 001 », le saut se produira vers l'adresse 08H, et ainsi de suite jusqu'à « 111 », qui provoque un saut à l'adresse 38H. L'instruction RST est très efficace, en termes de vitesse, puisqu'elle n'occupe qu'un seul octet. Cependant, elle n'effectue de branchements que vers huit adresses, situées en page 0, et séparées les unes des autres par seulement huit octets. Cette instruction est un héritage du 8080, où elle était utilisée intensivement pour les interruptions. Nous y reviendrons au chapitre consacré aux interruptions. Signalons toutefois que cette instruction peut être destinée à bien d'autres usages, et pourrait être assimilée à une éventuelle instruction spécialisée d'appel de sous-programme.

Exemples de sous-programmes

La plupart des programmes de cet ouvrage sont généralement écrits sous forme de sous-programmes. Par exemple, le programme de multiplication est destiné à être utilisé par de nombreuses parties d'un programme. Pour en faciliter et clarifier le développement, il serait intéressant de définir un sous-programme, dont le nom serait, par exemple, MULT. A la fin duquel il suffirait simplement d'ajouter l'instruction RET.

Exercice 3.32 : *Si MULT est utilisé en tant que sous-programme, existe-t-il un risque de « destruction » de certains registres ou de certains indicateurs ?*

Récursivité

La récursivité indique qu'un programme s'appelle lui-même. Si vous avez compris le mécanisme de fonctionnement des sous-programmes, vous devriez maintenant être capable de répondre à la question suivante :

Exercice 3.33 : *Est-il légal de laisser un sous-programme s'appeler lui-même ? (Autrement dit, tout fonctionnera-t-il convenablement lorsqu'un sous-programme s'appellera lui-même ? Si vous hésitez, dessinez la pile, et remplissez-là avec les adresses successives. Ensuite, regardez les registres et la mémoire (voir exercice 3.18) et dites si un problème se pose.*

Les interruptions seront traitées au chapitre consacré aux entrées-sorties (chapitre 6). Toutes les instructions de retour tiennent sur un octet, toutes les instructions d'appel (sauf RST) sur trois octets.

Exercice 3.34 : *Prenez connaissance, au chapitre suivant, des temps d'exécution des instructions CALL et RET. Pourquoi le retour est-il beaucoup plus rapide que l'appel ? (Conseil : si la réponse n'est pas évidente, examinez à nouveau l'utilisation de la pile dans le mécanisme des sous-programmes, et les nombres d'octets respectifs occupés par les instructions CALL et RET).*

Paramètres de sous-programmes

Un sous-programme est généralement appelé pour travailler sur des données. Par exemple, dans le cas de la multiplication, il s'agira de transmettre deux nombres au sous-programme, qui en effectuera le produit. Nous avons vu que le sous-programme s'attend à trouver le multiplicande et le multiplicateur à des adresses mémoires données. C'est une méthode de passage de paramètres, en mémoire. Il en existe deux autres. Trois possibilités, donc :

1. par les registres
2. par la mémoire
3. par la pile.

La solution des registres est avantageuse, car elle permet de faire l'économie d'un emplacement mémoire fixe : le sous-programme est indépendant de son emplacement en mémoire. Il faut, toutefois, disposer d'un nombre de registres suffisant. Si un emplacement mémoire fixe est utilisé, tout autre utilisateur du sous-programme devra veiller à utiliser la même convention, et à vérifier la disponibilité de cet emplacement mémoire (voir l'exercice 3.19). Dans de nombreux cas, un bloc de mémoire est réservé uniquement au passage des paramètres aux sous-programmes.

L'utilisation de la mémoire présente l'avantage d'une plus grande flexibilité (plus de données), mais engendre une performance médiocre. En outre, le sous-programme est lié à un emplacement fixe en mémoire.

Le dépôt des paramètres *sur la pile* offre le même avantage que l'utilisation des registres : indépendance vis-à-vis de l'emplacement mémoire. Le sous-programme sait simplement qu'il est supposé recevoir, disons, deux paramètres, qui seront stockés au sommet de la pile. Naturellement, il y a aussi des inconvénients : la pile est remplie de données, et le nombre de niveaux d'appels de sous-programmes s'en trouve réduit. Les manipulations de pile s'en trouvent compliquées à tel point qu'il faudra parfois recourir à plusieurs piles.

Le choix de la méthode appartient au programmeur. En général, il est préférable de retarder le plus longtemps possible l'astreinte de l'implantation physique en mémoire.

Si les registres sont indisponibles, la pile constitue une solution possible. Cependant, lorsqu'une grande quantité d'informations doit être communiquée à un sous-programme, il peut arriver que cette information réside en mémoire. Une manière élégante de contourner le problème du passage d'un bloc de données est, tout simplement, de transmettre un pointeur sur cette information. Un *pointeur* est l'adresse de début du bloc. Il peut être

```
         A=00 BC=0000 DE=0000 HL=0000 S=0300 P=0100 0100' LD   BC,(0200)
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0200')
         A=00 BC=0003 DE=0000 HL=0000 S=0300 P=0104 0104' LD   B,08
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
         A=00 BC=0803 DE=0000 HL=0000 S=0300 P=0106 0106' LD   DE,(0202)
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0202')
         A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010A 010A' LD   D,00
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
         A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010C 010C' LD   HL,0000
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0000')
         A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    C    A=00 BC=0801 DE=0005 HL=0000 S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0114')
    C    A=00 BC=0801 DE=0005 HL=0000 S=0300 P=0113 0113' ADD  HL,DE
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
         A=00 BC=0801 DE=0005 HL=0005 S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  V      A=00 BC=0801 DE=000A HL=0005 S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0801 DE=000A HL=0005 S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
   N     A=00 BC=0701 DE=000A HL=0005 S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (010F')
   N     A=00 BC=0701 DE=000A HL=0005 S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V C    A=00 BC=0700 DE=000A HL=0005 S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0114')
Z V C    A=00 BC=0700 DE=000A HL=0005 S=0300 P=0113 0113' ADD  HL,DE
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0700 DE=000A HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  V      A=00 BC=0700 DE=0014 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0700 DE=0014 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
   N     A=00 BC=0600 DE=0014 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (010F')
   N     A=00 BC=0600 DE=0014 HL=000F S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0600 DE=0014 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0114')
Z V      A=00 BC=0600 DE=0014 HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  V      A=00 BC=0600 DE=0028 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0600 DE=0028 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
   N     A=00 BC=0500 DE=0028 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (010F')
   N     A=00 BC=0500 DE=0028 HL=000F S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0500 DE=0028 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0114')
Z V      A=00 BC=0500 DE=0028 HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  V      A=00 BC=0500 DE=0050 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0500 DE=0050 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
   N     A=00 BC=0400 DE=0050 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (010F')
   N     A=00 BC=0400 DE=0050 HL=000F S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
```

Figure 3.39. — Multiplication : trace complète

```
Z V      A=00 BC=0400 DE=0050 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0114')
Z V      A=00 BC=0400 DE=0050 HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
S V      A=00 BC=0400 DE=00A0 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0400 DE=00A0 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    N    A=00 BC=0300 DE=00A0 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (010F')
    N    A=00 BC=0300 DE=00A0 HL=000F S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0300 DE=00A0 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0114')
Z V      A=00 BC=0300 DE=00A0 HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
      C  A=00 BC=0300 DE=0040 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
         A=00 BC=0300 DE=0140 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    N    A=00 BC=0200 DE=0140 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (010F')
    N    A=00 BC=0200 DE=0140 HL=000F S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0200 DE=0140 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0114')
Z V      A=00 BC=0200 DE=0140 HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
S        A=00 BC=0200 DE=0180 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
         A=00 BC=0200 DE=0280 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    N    A=00 BC=0100 DE=0280 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (010F')
    N    A=00 BC=0100 DE=0280 HL=000F S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V      A=00 BC=0100 DE=0280 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0114')
Z V      A=00 BC=0100 DE=0280 HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V C    A=00 BC=0100 DE=0200 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  V      A=00 BC=0100 DE=0500 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z   N    A=00 BC=0000 DE=0500 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (010F')
Z   N    A=00 BC=0000 DE=0500 HL=000F S=0300 P=011C 011C' LD   (0204),HL
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0204')
Z   N    A=00 BC=0000 DE=0500 HL=000F S=0300 P=011F 011F' NOP
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
```

Figure 3.39. — Multiplication : trace complète (suite)

transmis dans un registre, sur la pile (deux emplacements sont utilisés pour ranger une adresse sur 16 bits), ou à une adresse mémoire donnée.

Si aucune de ces solutions n'est applicable, il faudra convenir, avec le sous-programme, d'un emplacement mémoire donné (une « boîte aux lettres ») pour y déposer les informations.

Exercice 3.35 : *Laquelle de ces trois méthodes est la meilleure pour la récursivité ?*

Bibliothèque de sous-programmes

La structuration des parties de programmes en sous-programmes identifiables présente un intérêt considérable : ces programmes peuvent être mis au point de manière indépendante, et avoir un nom. Utilisés par d'autres zones du programme, ils peuvent être partagés, et constituer une bibliothèque de sous-programmes. Il n'y a toutefois aucune panacée universelle, en programmation. L'utilisation systématique des sous-programmes, pour n'importe quel groupe d'instructions effectuant une fonction, peut entraîner une efficacité médiocre. Le programmeur habile devra s'efforcer de peser les avantages et les inconvénients.

CONCLUSION

Ce chapitre a exposé la manière dont l'information est traitée par les instructions, au sein du Z80. Des algorithmes de complexité croissante ont été introduits et traduits en programmes. Les principaux types d'instructions ont été utilisés et expliqués.

Des structures importantes telles que les boucles, les piles et les sous-programmes ont été définies.

Vous devriez maintenant avoir assimilé les bases de la programmation et les principales techniques utilisées dans les applications standard. Nous allons maintenant étudier le jeu d'instructions disponible sur le Z80.

RÉPONSES A L'EXERCICE 3.18 (MULTIPLICATION) :

```
  CROMEMCO CDOS Z80 ASSEMBLER version 02.15                PAGE 0001

0000              0001          ORG     0100H
      (0200)      0002 MPRAD    DL      0200H
      (0202)      0003 MPDAD    DL      0202H
      (0204)      0004 RESAD    DL      0204H
                  0005 ;
0100  ED4B0002    0006 MP488    LD      BC,(MPRAD)     ;LOAD MULTIPLIER INTO C
0104  0608        0007          LD      B,8            ;B IS BIT COUNTER
0106  ED5B0202    0008          LD      DE,(MPDAD)     ;LOAD MUTIPLICAND INTO E
010A  1600        0009          LD      D,0            ;CLEAR D
010C  210000      0010          LD      HL,0           ;SET RESULT TO 0
010F  CB39        0011 MULT     SRL     C              ;SHIFT MULTIPLIER BIT INTO CARRY
0111  3001        0012          JR      NC,NOADD       ;TEST CARRY
0113  19          0013          ADD     HL,DE          ;ADD MPD TO RESULT
0114  CB23        0014 NOADD    SLA     E              ;SHIFT MPD LEFT
0116  CB12        0015          RL      D              ;SAVE BIT IN D
0118  05          0016          DEC     B              ;DECREMENT SHIFT COUNTER
0119  C20F01      0017          JP      NZ,MULT        ;DO IT AGAIN IF COUNTER <> 0
011C  220402      0018          LD      (RESAD),HL     ;STORE RESULT
011F  (0000)      0019          END

Errors            0
```

Figure 3.40. — Le programme de multiplication (hex)

ETIQUETTE	INSTRUCTION	B	C	C (REPORT)	D	E	H	L
		00	00	0	00	00	00	00
MULT 88	LD BC,(0200)	00	03	0	00	00	00	00
	LD B, 08	08	03	0	00	00	00	00
	LD DE,(0202)	08	03	0	00	05	00	00
	LD D, 00	08	03	0	00	05	00	00
	LD HL,0000	08	03	0	00	05	00	00
MULT	SRL C	08	01	1	00	05	00	00
	JR NC,0114	08	01	1	00	05	00	00
	ADD HL,DE	08	01	0	00	05	00	05
NOADD	SLA E	08	01	0	00	0A	00	05
	RL D	08	01	0	00	0A	00	05
	DEC B	07	01	0	00	0A	00	05
	JP NZ,010F	07	01	0	00	0A	00	05
MULT	SRL C	07	00	1	00	0A	00	05
	JR NC,0114	07	00	1	00	0A	00	05
	ADD HL,DE	07	00	0	00	0A	00	0F
NOADD	SLA E	07	00	0	00	14	00	0F
	RL D	07	00	0	00	14	00	0F
	DEC B	06	00	0	00	14	00	0F
	JP NZ,010F	06	00	0	00	14	00	0F

Figure 3.41. — Deux passages dans la boucle

4

LE JEU D'INSTRUCTIONS DU Z80

INTRODUCTION

Dans ce chapitre, nous commencerons par analyser les différentes classes d'instructions disponibles sur un ordinateur d'usage général ; ensuite, nous analyserons, une par une, toutes les instructions du Z80, en expliquant en détail leur fonction, leur mode d'affectation des indicateurs et la manière dont elles peuvent être utilisées avec les divers modes d'adressage. Ces derniers seront étudiés au chapitre 5.

LES CLASSES D'INSTRUCTIONS

Les instructions peuvent être classées de différentes manières, indépendamment de tout standard. Nous distinguerons cinq grandes catégories :

1. Transfert de données
2. Traitement de données
3. Tests et branchements
4. Entrées-sorties
5. Contrôle

Examinons successivement chaque classe d'instructions

Transfert de données

Les instructions correspondantes transfèrent les données entre registres, entre un registre et la mémoire, ou entre un registre et un organe d'entrée-sortie. Il peut se trouver des instructions de transfert spécialisées, associées à des registres dotés d'un rôle spécifique. Par exemple, l'utilisation efficace d'une pile recourt à des opérations d'empilement et de dépilement. Ces opérations transmettent une donnée d'un mot entre le sommet de la pile et l'accumulateur, et mettent à jour le registre pointeur de pile. Le tout en une seule instruction.

Traitement de données

Les instructions de traitement de données sont de cinq sortes :

1. Opérations arithmétiques (telles que plus/moins)
2. Manipulation de bits (mise à zéro ou à un)
3. Incrémentation et décrémentation
4. Opérations logiques (telles que ET, OU, OU exclusif)
5. Opérations de déplacement et de décalage (telles que rotation, décalage).

A noter que l'efficacité du traitement de données commande de disposer d'instructions arithmétiques puissantes, telles que la multiplication et la division. Malheureusement, celles-ci ne sont pas disponibles sur la plupart des microprocesseurs. Il conviendrait aussi de disposer d'instructions de décalage et de déplacement puissantes, telles que le décalage de n bits ou l'échange de quartets (permutation des moitiés droite et gauche d'un octet). Mais elles non plus ne sont habituellement pas disponibles sur microprocesseurs.

Avant d'examiner les instructions du Z80 soulignons, une nouvelle fois, la différence entre un *décalage* et une *rotation.* Le décalage déplace le contenu d'un registre, ou d'une case mémoire, d'un bit vers la droite ou vers la gauche. Le bit ainsi chassé du registre est placé dans le bit de report. Le bit entrant par l'autre extrémité est un « 0 », sauf dans le cas du « décalage arithmétique à droite », où le bit de poids fort est dupliqué.

Dans le cas de la rotation, le bit sortant est également placé dans le bit de report. Par contre, la valeur précédente du bit de report est utilisée comme bit entrant. Ceci correspondant à une rotation sur 9 bits. Il est souvent souhaitable de mettre en œuvre une vraie rotation sur 8 bits : le bit sortant d'un côté entrant alors de l'autre côté. C'est malheureusement impossible sur la majorité des microprocesseurs, mais pas sur le Z80 (voir figure 4.1.)

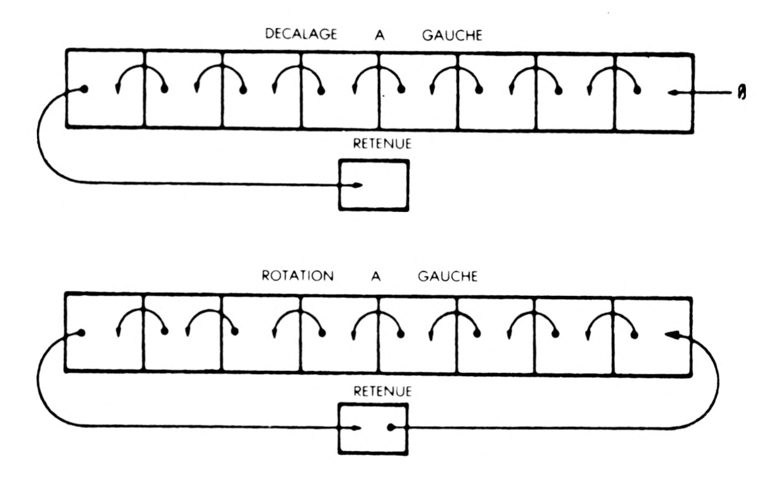


Figure 4.1. — Décalage et rotation

Enfin, pour décaler un mot vers la droite, il est commode de disposer d'un type supplémentaire de décalage, appelé « extension du signe », ou « décalage arithmétique à droite ». Pour opérer sur des nombres en complément à deux, et notamment pour écrire des routines de calcul en virgule flottante, il est souvent nécessaire de décaler un nombre négatif vers la droite. Dans ce cas, le bit qui entre par la gauche doit être un « 1 » (le signe doit être répété chaque fois qu'un décalage a lieu). C'est le décalage arithmétique à droite.

Test et branchement

Les instructions de test permettent de vérifier que les bits des registres désignés sont à l'état « 0 » ou « 1 », ou une combinaison de « 0 » et de « 1 ». Il doit être, au minimum, possible de tester le registre d'indicateurs. Il est alors souhaitable de disposer dans ce registre d'autant d'indicateurs que possible. De plus, il est commode de pouvoir y déceler en une seule instruction une configuration de bits. Enfin la possibilité de tester n'importe quel bit de n'importe quel registre, et de comparer les valeurs respectives de deux registres, (plus grand, plus petit, égal) est souhaitable. Les instructions de test des microprocesseurs se limitent, en général, au test d'un seul bit du registre d'indicateurs. Le Z80 offre, à cet égard, des perspectives supérieures à beaucoup d'autres.

Les instructions de saut disponibles sont, en général, de trois types :

1. le saut, indiquant une adresse complète de 16 bits,
2. le saut relatif, généralement limité à un déplacement codé sur 8 bits,
3. l'appel, utilisé pour les sous-programmes.

Il est intéressant de disposer de branchements à deux, voire trois voies, selon, par exemple, que le résultat d'une comparaison indique « plus grand », « plus petit » ou « égal ». Autre commodité : les instructions permettant de sauter quelques instructions, en avant ou en arrière. Mais ces instructions de saut sont équivalentes à des branchements. Enfin, au terme de la plupart des boucles, une opération d'incrémentation, ou de décrémentation, a lieu, suivie d'un test et d'un branchement. L'existence d'une instruction unique effectuant l'incrémentation [ou la décrémentation], le test puis le saut, est un avantage important pour la mise en œuvre efficace des boucles. La plupart des microprocesseurs n'en disposent pas : ils n'offrent que des branchements simples, combinés à des tests simples. Tout cela complique naturellement la programmation, et en réduit l'efficacité. Le Z80 dispose d'une instruction « décrémenter et se brancher », qui n'est cependant mise en œuvre que si un registre particulier (B) est nul

Entrées-sorties

Les instructions d'entrée-sortie sont spécialisées dans la gestion des organes correspondants. La majorité des microprocesseurs 8 bits utilisent

notamment la projection des entrées-sorties sur l'espace mémoire : les organes d'E/S sont connectés au bus d'adresses et adressés exactement comme les circuits mémoires. Ils apparaissent au programmeur comme des emplacements mémoire. Toutes les opérations impliquant la mémoire nécessitent normalement trois octets. Elles sont donc lentes. Pour gérer efficacement les entrées-sorties dans un tel environnement, il est souhaitable de disposer d'un mécanisme d'adressage court, de sorte que les organes périphériques exigeant une gestion rapide résident en page 0. Pourtant, lorsque l'adressage en page 0 existe, il est généralement utilisé par de la mémoire RAM. Comme le 8080, le Z80, dispose d'instructions spécialisées d'entrée-sortie. Il permet ainsi au concepteur d'utiliser les deux méthodes. Les organes d'entrée-sortie pourront donc, au choix, être adressés comme des éléments de mémoire, ou comme des éléments d'entrée-sortie, en utilisant les instructions d'entrée-sortie.

Ces dernières seront décrites plus avant dans ce chapitre.

Instructions de contrôle

Les instructions de contrôle fournissent des signaux de contrôle, et sont susceptibles de suspendre ou d'interrompre un programme. Elles peuvent aussi fonctionner à la manière d'un point d'arrêt, ou d'une interruption simulée (les interruptions seront décrites au chapitre 6, consacré aux techniques d'entrée-sortie).

LE JEU D'INSTRUCTIONS DU Z80

Introduction

Le microprocesseur Z80 a été conçu, à la fois, pour être compatible avec le 8080, et pour offrir des possibilités supplémentaires. Il résulte de cette philosophie de conception que le Z80 dispose de toutes les instructions du 8080, et de certaines instructions supplémentaires. Compte-tenu du nombre limité de bits disponibles dans un code opératoire de 8 bits, il est permis de se demander comment les concepteurs du Z80 ont réussi à implanter de nombreux codes opératoires supplémentaires. Ils l'ont fait en utilisant les rares codes inutilisés du 8080, et en ajoutant un octet supplémentaire au code opératoire, pour les opérations indexées. C'est pourquoi certaines instructions du Z80 occupent jusqu'à cinq octets en mémoire.

Il faut garder à l'esprit qu'un programme peut-être écrit de bien des façons. Une connaissance et une compréhension totales du jeu d'instructions est indispensable pour parvenir à une programmation efficace. Cependant, le débutant en programmation pourra se dispenser d'écrire des programmes optimisés. Lisant ce chapitre pour la première fois, il pourra négliger de se souvenir de toutes les instructions. Il aura, par contre, tout intérêt à se rappeler les catégories d'instructions, et à étudier les exemples caractéristiques. Pour écrire des programmes, le lecteur devra donc

consulter la description du jeu d'instructions du Z80, et choisir parmi elles les mieux adaptées à ses besoins. Les diverses instructions du Z80 seront passées en revue, dans cette section, avec l'objectif de les schématiser, et de les regrouper en catégories logiques. Le lecteur intéressé à explorer les possibilités des diverses instructions est invité à se reporter à leurs descriptions individuelles.

Examinons les possibilités offertes par le Z80 selon les cinq classes d'instructions définies au début de ce chapitre.

Instructions de transfert de données

Les instructions de transfert de données du Z80 peuvent elles-mêmes être classées en quatre catégories : transferts de 8 bits, transferts de 16 bits, opérations avec la pile, et transferts par blocs.

Passons les en revue.

Transfert sur 8 bits

Tous les transferts de 8 bits se font au moyen d'instructions de chargement. Le format en est :

LD destination, source

Par exemple, l'accumulateur A peut être chargé à partir du registre B, en utilisant l'instruction :

LD A, B

Des transferts directs peuvent être accomplis entre n'importe lesquels des registres de travail (ABCDEHL).

Pour charger un registre de travail quelconque, à partir d'une case mémoire, il est nécessaire de charger au préalable l'adresse de cette case mémoire dans un registre double, tel que la paire H et L. Sauf dans le cas de l'accumulateur.

Par exemple, pour charger le registre C à partir de la case mémoire 1234, le registre formé par H et L devra, au préalable, être chargé avec la valeur « 1234 ». (Une instruction de chargement sur 16 bits devra être utilisée. Ce type d'instruction sera décrit au paragraphe suivant).

Ensuite, l'instruction LD C, (HL) accomplira l'action voulue.

Le cas de l'accumulateur constitue une exception. Il peut être chargé directement, depuis n'importe quelle case mémoire désignée. C'est ce qu'on appelle le mode d'adressage étendu. Par exemple, pour charger l'accumulateur avec le contenu de la case mémoire 1234, nous utiliserons l'instruction :

LD A, (1234 H). (remarquons l'utilisation des parenthèses pour désigner

« le contenu de »). Cette instruction est rangée en mémoire de la façon suivante :

Adresse	PC	: 3A	(code opération)
	PC + 1	: 34	(moitié de poids faible de l'adresse)
	PC + 2	: 12	(moitié de poids fort de l'adresse)

A noter que l'adresse est rangée, dans l'instruction elle-même, en « ordre renversé ».

3A	Adresse poids faible	Adresse poids fort

Tous les registres de travail peuvent aussi être chargés avec n'importe quelle valeur sur 8 bits, appelée « littéral » et contenue dans le second octet de l'instruction. C'est l'adressage immédiat. En voici un exemple :

LD E, 12 H

qui charge le règistre E avec la valeur hexadécimale 12.

En mémoire, l'instruction apparaît ainsi :

PC	: 1E	(code opération)
PC + 1	: 12	(opérande littéral)

Après exécution de cette instruction, le registre E contiendra l'opérande immédiat, ou valeur littérale.

Il existe aussi un mode *d'adressage indexé,* permettant de charger le contenu d'un registre. Il sera décrit, dans sa totalité, au chapitre suivant [les techniques d'adressage]. Diverses possibilités permettent encore de charger l'un ou l'autre des registres. Un tableau recensant toutes ces possibilités est établi à la figure 4.2. (1)
Les zones teintées de gris montrent les instructions communes au Z80 et au 8080 A.

Transferts sur 16 bits

N'importe quel registre 16 bits [BC, DE, HL, SP, IX, IY] peut être chargé avec un opérande immédiat de 16 bits, depuis une adresse mémoire désignée (*adressage étendu),* ou depuis le sommet de la pile, c'est-à-dire depuis l'adresse contenue dans SP. Le contenu de ces registres doubles peut être rangé, de la même manière, à une adresse mémoire désignée, ou sur le sommet de la pile. De plus, le registre SP peut être chargé à partir de HL, IX et IY. Cette particularité permet de créer plusieurs piles. La paire de registres AF peut, elle aussi, être chargée au sommet de la pile.

(1) Tableau fourni par ZILOG Inc.

		Source															
		Implicite		Registre							Registre indirect			Indexé		Adres étendu	IMME.
		I	R	A	B	C	D	E	H	L	(HL)	(BC)	(DE)	(IX + d)	(IY + d)	(nn)	n
Destination — Registre	A	ED 57	ED 5F	7F	78	79	7A	7B	7C	7D	7E	0A	1A	DD 7E d	FD 7E d	3A n n	3E n
	B			47	40	41	42	43	44	45	46			DD 46 d	FD 46 d		06 n
	C			4F	48	49	4A	4B	4C	4D	4E			DD 4E d	FD 4E d		0E n
	D			57	50	51	52	53	54	56	56			DD 56 d	FD 56 d		16 n
	E			5F	58	59	5A	5B	5C	5D	5E			DD 5E d	FD 5E d		1E n
	H			67	60	61	62	63	64	65	66			DD 66 d	FD 66 d		26 n
	L			6F	68	69	6A	6B	6C	6D	6E			DD 6E d	FD 6E d		2E n
Registre Indirect	(HL)			77	70	71	72	73	74	75							36 n
	(BC)			02													
	(DE)			12													
Indexé	(IX+d)			DD 77 d	DD 70 d	DD 71 d	DD 72 d	DD 73 d	DD 74 d	DD 75 d							DD 36 d n
	(IY+d)			FD 77 d	FD 70 d	FD 71 d	FD 72 d	FD 73 d	FD 74 d	FD 75 d							FD 36 d n
Adressage étendu	(nn)			32 n n													
Implicite	I			ED 47													
	R			ED 4F													

Figure 4.2. — Groupe des chargements 8 bits « LD »

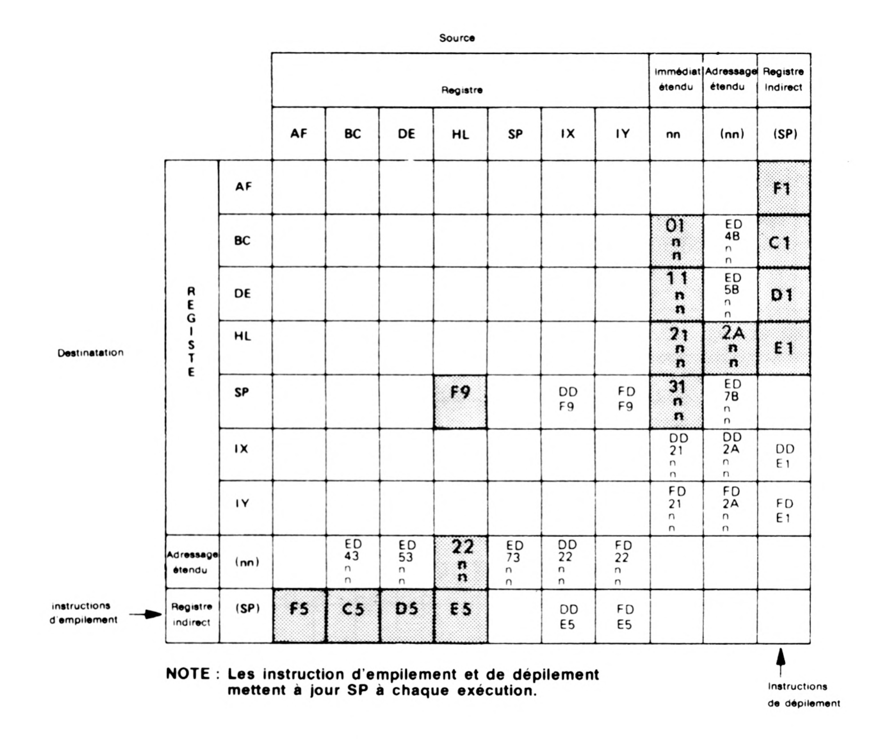
Le tableau recensant toutes les possibilités est développé à la figure 4.3. Les opérations sur la pile d'empilement [ou de dépilement] font partie des transferts sur 16 bits. Toutes les opérations sur la pile, transfèrent le contenu d'un registre double vers [ou depuis] la pile. Remarquons que nulle instruction d'empilement ou de dépilement ne permet de sauvegarder individuellement un registre de 8 bits.

L'empilement ou le dépilement de deux octets est toujours exécuté sur un registre double : AF, BC, DE, HL, IX, IY (voir la dernière ligne et la colonne de droite de la figure 4.3).

Avec AF, BC, DE, ou HL, l'instruction ne nécessite qu'un seul octet, et procure une bonne efficacité. Supposons, par exemple, que le pointeur de pile SP contienne la valeur « 0100 », et que soit exécutée l'instruction

PUSH AF

Lorsque le contenu de la paire de registres est empilé, le pointeur de pile SP est d'abord décrémenté. Le contenu du registre A est ensuite déposé au



Destinatation		Source: Registre AF	BC	DE	HL	SP	IX	IY	Immédiat étendu nn	Adressage étendu (nn)	Registre Indirect (SP)
REGISTE	AF										F1
REGISTE	BC								01 n n	ED 4B n n	C1
REGISTE	DE								11 n n	ED 5B n n	D1
REGISTE	HL								21 n n	2A n n	E1
REGISTE	SP				F9		DD F9	FD F9	31 n n	ED 7B n n	
REGISTE	IX								DD 21 n n	DD 2A n n	DD E1
REGISTE	IY								FD 21 n n	FD 2A n n	FD E1
Adressage étendu	(nn)		ED 43 n n	ED 53 n n	22 n n	ED 73 n n	DD 22 n n	FD 22 n n			
Registre indirect (instructions d'empilement)	(SP)	F5	C5	D5	E5		DD E5	FD E5			

Figure 4.3. — Groupe des chargements 16 bits « LD », « PUSH » et « POP »

sommet de la pile. SP est à nouveau décrémenté, et le contenu de F est déposé au sommet de la pile. A la fin du transfert, SP pointe sur l'élément situé au sommet de la pile : dans notre exemple, la valeur de F.

Il est important de ne pas perdre de vue que, dans le cas du Z80, le pointeur de pile SP pointe sur *le sommet de la pile,* et que SP est *décrémenté* chaque fois qu'un registre double est empilé. Avec d'autres processeurs, d'autres conventions sont souvent utilisées. D'où un risque non-négligeable de confusion.

Instructions d'échange

Un mnémonique spécial EX est, en outre, réservé aux opérations d'échange. EX n'est pas un simple transfert de données, mais un transfert double. Il change réellement les contenus de deux emplacements désignés, et peut être utilisé pour échanger le sommet de la pile avec HL, IX, IY, ou pour intervertir les contenus de DE et HL, ou de AF et AF' (on se souvient que AF' désigne l'autre paire de registres AF disponible dans le Z80).

Enfin, une instruction spéciale EXX permet d'échanger les contenus de BC, DE et HL avec les contenus des registres correspondants dans le second jeu.

Les échanges possibles sont résumés par la figure 4.4.

		Adressage implicite				
		AF'	BC', DE' & HL'	HL	IX	IY
Implicite	AF	08				
	BC, DE & HL		D9			
	DE			EB		
Registre indirect	(SP)			E3	DD E3	FD E3

Figure 4.4. — Echanges EX et EXX

Instructions de transfert de blocs

Les instructions de transfert de blocs opèrent le transfert d'un bloc de données, et non pas seulement d'un ou de deux octets. Leur implantation par le constructeur est plus difficile que celle de la plupart des autres instructions. Les microprocesseurs n'en disposent généralement pas. Elles permettent une programmation commode, et sont susceptibles d'améliorer les performances d'un programme, surtout pendant les opérations d'entrée-sortie. Leur utilisation et leurs avantages seront montrés au fil de ce livre. Certaines instructions de transfert automatique de blocs sont disponibles sur le Z80. Elles utilisent des conventions spécifiques.

Toutes nécessitent l'utilisation de trois registres doubles : BC, DE et HL ; BC sert de compteur de 16 bits, ce qui signifie que le déplacement automatique est possible jusqu'à 2^{16} = 64 K octets ; HL est utilisé comme pointeur-source, et peut désigner n'importe quel emplacement mémoire ; DE est le pointeur destinatination : il peut pointer n'importe quel emplacement mémoire.

Quatre instructions de transfert de blocs sont fournies :

LDD, LDDR, LDI, LDIR

Chacune d'elles décrémente le registre BC [utilisé comme compteur], à chaque transfert. Deux d'entre elles, LDD et LDDR, décrémentent les registres DE et HL [utilisés comme pointeurs], et les deux autres, LDI et

LDIR, incrémentent DE et HL. Dans chaque groupe, la lettre R, à la fin du mnémonique, indique une répétition automatique. Examinons ces instructions.

LDI signifie « charger et incrémenter » (en anglais : « Load and increment »). Cette instruction transfère un octet de l'emplacement mémoire pointé par H et L, vers celui pointé par D et E. Elle décrémente aussi BC, et incrémente automatiquement HL et DE, de façon que tous les registres doubles contiennent les valeurs appropriées à l'exécution éventuelle du transfert suivant.

LDIR signifie « charger, incrémenter et répéter » (en anglais : « load, increment and repeat »). Autrement dit : exécuter répétitivement LDI, jusqu'à ce que le compteur BC atteigne la valeur « 0 ». Cette instruction est utilisée pour déplacer automatiquement un bloc de données contiguës, d'une zone mémoire vers une autre.

LDD et LDDR opèrent de la même façon, à cette différence près que les pointeurs sont *décrémentés,* au lieu d'être incrémentés. Le transfert a lieu en commençant par l'adresse la *plus haute* du bloc, au lieu de la plus basse. Les effets de ces quatre instructions sont résumés à la figure 4.5

			Source	
			Registre indirect	
			(HL)	
Destination	Registre indirect	(DE)	ED A0	'LDI' charger (DE) ← (HL) Inc HL & DE, Dec BC
			ED B0	'LDIR' charger (DE) ← (HL) Inc HL & DE, Dec BC, répéter jusqu'à BC = 0
			ED A8	'LDD' — charger (DE) ← (HL) Dec HL & DE, Dec BC
			ED B8	'LDDR' — charger (DE) ← (HL) Dec HL & DE, Dec BC, répéter jusqu'à BC ∅ 0

Reg HL pointe la source
Reg DE pointe la destination
Reg BC sert de compte d'octets.

Figure 4.5. — Groupe des transferts de bloc

Des instructions similaires existent pour CP (comparer). Elles automatisent les recherches par bloc et sont résumées à la figure 4-6.

Zone de recherche Registre indirect (HL)	
ED A1	'CPI' Inc HL, Dec BC
ED B1	'CPIR' Inc HL, Dec BC répéter jusqu'à BC = 0 ou élément trouvé
ED A9	'CPD' Dec HL & BC
ED B9	'CPDR' Dec HL & BC répéter jusqu'à BC = 0 ou élément trouvé

HL pointe sur l'emplacement mémoire qui doit être comparé à l'accumulateur
BC sert de compte d'octets

Figure 4.6. — Groupe de recherche par bloc

Instructions de traitement de données

Arithmétique

Deux principales instructions arithmériques sont fournies : addition et soustraction. Elles ont été utilisées, de façon intensive, au chapitre précédent. Nous avons distingué deux types d'addition, avec et sans report : respectivement, ADD et ADC ; et également deux types de soustraction, avec et sans report : SUB et SBC.

De plus, on dénombre trois instructions spéciales : DAA, CPL et NEG.

Nous avons eu recours à l'instruction d'ajustement décimal de l'accumulateur, DAA, pour réaliser des opérations DCB. Elle est normalement utilisée pour chaque addition ou soustraction DCB. Deux instructions de complémentation sont aussi disponibles. CPL complémente à un l'accumulateur, et NEG en prend l'opposé, c'est-à-dire le complément à deux.

Toutes les instructions précédentes opèrent sur des données de huit bits. Les opérations sur 16 bits sont plus restreintes. ADD, ADC et SBC ne sont possibles que sur certains registres (voir figure 4.8).

Enfin, des instructions d'incrémentation et de décrémentation existent pour tous les registres, qu'ils soient de 8 ou de 16 bits. Elles sont recensées aux figures 4.7 (opérations sur 8 bits), et 4.8 (opérations sur 16 bits).

Source

	Registre							Registre indirect	Indexé		Imméd.
	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)	n
Addition ADD	87	80	81	82	83	84	85	86	DD 86 d	FD 86 d	C6 n
Add. avec le report ADC	8F	88	89	8A	8B	8C	8D	8E	DD 8E d	FD 8E d	CE n
Soustraction SUB	97	90	91	92	93	94	95	96	DD 96 d	FD 96 d	D6 n
Soust. avec le report SBC	9F	98	99	9A	9B	9C	9D	9F	DD 9E d	FD 9E d	DE n
Et logique AND	A7	A0	A1	A2	A3	A4	A5	A6	DD A6 d	FD A6 d	E6 n
Ou exclusif XOR	AF	A8	A9	AA	AB	AC	AD	AE	DD AE d	FD AE d	EE n
Ou logique OR	B7	B0	B1	B2	B3	B4	B5	B6	DD B6 d	FD B6 d	F6 n
Comparaison CP	BF	B8	B9	BA	BB	BC	BD	BE	DD BE d	FD BE d	FE n
Incrémentation INC	3C	04	0C	14	1C	24	2C	34	DD 34 d	FD 34 d	
Décrémentation DEC	3D	05	0D	15	1D	25	2D	35	DD 35 d	FD 35 d	

Figure 4.7. — Arithmétique et logique huit bits

En général, toutes les opérations arithmétiques modifient certains indicateurs. Leurs effets sont décrits, en totalité, dans la seconde partie de ce chapitre. Il est toutefois important de noter que les instructions INC et DEC, lorsqu'elles portent sur des registres doubles, ne modifient aucun indicateur. Ce détail doit être gardé à l'esprit. Conséquence : si un registre double atteint la valeur « 0 », par suite d'une incrémentation ou d'une décrémentation, le bit Z du registre d'indicateur F n'est pas positionné. Il est donc nécessaire, dans le programme, de tester explicitement la valeur [« 0 » ou non] du registre double.

Autre fait notable : les instructions ADC et SBC affectent toujours l'ensemble des indicateurs, ce qui ne signifie pas forcément que tous les indicateurs seront nécessairement différents, après l'exécution de l'instruction. C'est simplement une éventualité.

Source

Destination		BC	DE	HL	SP	IX	IY
Addition ADD	HL	09	19	29	39		
	IX	DD 09	DD 19		DD 39	DD 29	
	IY	FD 09	FD 19		FD 39		FD 29
Add. avec le report ADC	HL	ED 4A	ED 5A	ED 6A	ED 7A		
Soustraction avec le report SBC	HL	ED 42	ED 52	ED 62	ED 72		
Incrémentation INC		03	13	23	33	DD 23	FD 23
Décrémentation DEC		0B	1B	2B	3B	DD 2B	FD 2B

Figure 4.8. — Arithmétique seize bits

Logique

Il existe trois opérations logiques : AND (ET), OR (OU inclusif) et XOR (OU exclusif), plus une instruction de comparaison CP. Elles portent toutes, exclusivement, sur des données huit bits. Passons les en revue (un tableau recensant toutes les possibilités, et tous les codes opératoires, de ces instructions est inclus dans la figure 4.7).

AND (ET)

Chaque opération logique est caractérisée par une *table de vérité*, qui exprime la valeur logique du résultat, en fonction des entrées. La table de vérité de la fonction ET est la suivante :

0 ET 0 = 0
0 ET 1 = 0
1 ET 0 = 0
1 ET 1 = 1

ou

ET	0	1
0	0	0
1	0	1

L'opération ET se caractérise par le fait que la sortie vaut « 1 » si, et seulement si, les deux entrées valent « 1 ». En d'autres termes, si l'une des entrées vaut « 0 », il est certain que le résultat est « 0 ». Cette propriété permet de mettre à zéro un bit dans un mot. C'est ce qu'on appelle le « masquage ».

Une des utilisations importantes de l'instruction AND, est de mettre à zéro, ou de masquer, un ou plusieurs bits dans un mot. Supposons, par exemple, qu'il s'agisse de mettre à zéro les quatre bits situés à l'extrême-droite d'un mot. Le programme sera le suivant :

```
LD    A, MOT        MOT contient « 10101010 »
AND   11110000B     « 11110000 » est le masque
```

Supposons maintenant que MOT vaille « 10101010 ». Le résultat du programme sera de laisser « 10100000 » dans l'accumulateur. Le postfixe « B » est utilisé pour désigner une valeur binaire

Exercice 4.1 : *Écrire un programme de trois lignes qui mette à zéro les bits 1 et 6 de MOT.*

Exercice 4.2 : *Que se passe-t-il avec un masque égal à « 11111111 » ?*

OR (OU)

Cette instruction correspond à l'opération OU inclusif. Elle se caractérise par la table de vérité suivante :

0 OU 0 = 0
0 OU 1 = 1
1 OU 0 = 1
1 OU 1 = 1

ou

OU	0	1
0	0	1
1	1	1

Le OU logique se caractérise par le fait que, si l'une des entrées vaut « 1 », alors le résultat vaut « 1 ». Son utilisation la plus évidente consiste à mettre n'importe quel bit à « 1 ».

Mettons donc à 1 les quatre bits les plus à droite de MOT. Le programme s'écrit :

```
LD    A, MOT
OR    00001111B
```

Si MOT contient « 10101010 », la valeur finale de l'accumulateur sera « 10101111 ».

Exercice 4.3 : *Que se passerait-il si nous utilisions l'instruction OR 10101111B ?*

Exercice 4.4 : *Quel est l'effet d'un OU avec « FF » en hexadécimal ?*

XOR (OU exclusif)

XOR désigne le « OU exclusif », qui diffère du OU inclusif sur un point : le résultat vaut « 1 » si un, et un seulement, des opérandes est égal à « 1 ». Si les deux opérandes sont égaux à « 1 », le OU normal donne le résultat « 1 », et le OU exclusif le résultat « 0 ». Voici la table de vérité :

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

ou

XOR	0	1
0	0	1
1	1	0

Le OU exclusif est utilisé pour les comparaisons. Si un seul bit est différent, le OU exclusif des deux mots ne sera pas nul. En outre dans le cas du Z80, il peut être utilisé pour *complémenter* un mot, dans la mesure où il ne se trouve pas d'instruction de complémentation ailleurs que sur l'accumulateur. Ce résultat est obtenu en effectuant un OU exclusif, avec un mot tout à 1. Le programme s'écrit :

```
LD      A, MOT
XOR     11111111B
LD      r, A
```

où r désigne le registre.

Supposons que MOT contienne « 10101010 ». La valeur finale du registre sera « 01010101 ». Il est possible de vérifier qu'il s'agit là du complément de la valeur initiale.

Le OU exclusif peut être avantageusement utilisé pour inverser un bit.

Exercice 4.5 : *Quel est l'effet du OU exclusif d'un registre avec « 00 » en hexadécimal ?*

Opérations de décalage et de rotation

Précisons la différence entre les opérations de décalage et de rotation, représentées à la figure 4.9. Dans une opération de décalage, le contenu du registre est décalé d'un bit vers la gauche, ou vers la droite. Le bit qui sort du registre est dirigé vers le bit de report C, et le bit qui y entre est un zéro. L'opération a été expliquée au début du chapitre.

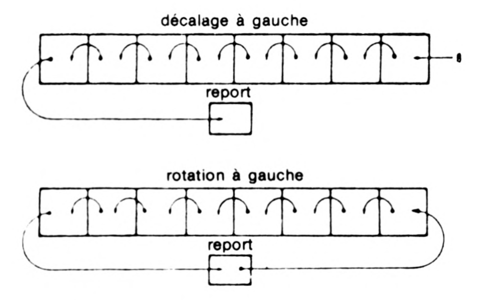
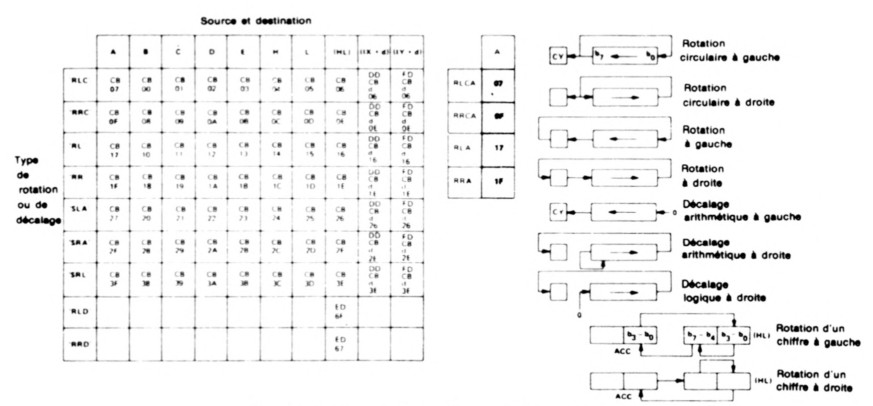


Figure 4.9. — Décalage et rotation

Il y a pourtant une exception : le décalage arithmétique à droite. Lors d'opérations sur des nombres en complément à deux, la valeur du bit le plus à gauche [le bit de signe] est « 1 » quand le nombre est négatif. Lorsqu'un nombre négatif est divisé par « 2 », par décalage vers la droite, il doit rester négatif, c'est dire que le bit situé le plus à gauche doit rester un « 1 ». La manœuvre est réalisée automatiquement par l'instruction de décalage arithmétique vers la droite, SRA, au cours de laquelle le bit qui entre dans le registre par la gauche est identique au bit de signe initial. Il s'agira d'un « 0 », si le bit le plus à gauche est « 0 », et d'un « 1 », si le bit le plus à gauche est « 1 ». Tout cela est schématiquement illustré sur la partie droite de la figure 4.10, qui énumère toutes les opérations de décalage et de rotation possibles.

Rotations

Une rotation diffère d'un décalage en ce que le bit qui entre dans le registre est celui qui sort, soit par l'autre extrémité du registre, soit du bit de report. Il existe deux types de rotation sur le Z80 : une rotation sur huit bits, et une autre sur neuf bits.



Source et destination

Type de rotation ou de décalage	A	B	C	D	E	H	L	(HL)	(IX + d)	(IY + d)
RLC	CB 07	CB 00	CB 01	CB 02	CB 03	CB 04	CB 05	CB 06	DD CB d 06	FD CB d 06
RRC	CB 0F	CB 08	CB 09	CB 0A	CB 0B	CB 0C	CB 0D	CB 0E	DD CB d 0E	FD CB d 0E
RL	CB 17	CB 10	CB 11	CB 12	CB 13	CB 14	CB 15	CB 16	DD CB d 16	FD CB d 16
RR	CB 1F	CB 18	CB 19	CB 1A	CB 1B	CB 1C	CB 1D	CB 1E	DD CB d 1E	FD CB d 1E
SLA	CB 27	CB 20	CB 21	CB 22	CB 23	CB 24	CB 25	CB 26	DD CB d 26	FD CB d 26
SRA	CB 2F	CB 28	CB 29	CB 2A	CB 2B	CB 2C	CB 2D	CB 2E	DD CB d 2E	FD CB d 2E
SRL	CB 3F	CB 38	CB 39	CB 3A	CB 3B	CB 3C	CB 3D	CB 3E	DD CB d 3E	FD CB d 3E
RLD								ED 6F		
RRD								ED 67		

	A
RLCA	07
RRCA	0F
RLA	17
RRA	1F

Figure 4.10. — Rotation et décalages

La rotation sur neuf bits est illustrée par la figure 4.11. Dans le cas de la rotation à droite, les huit bits du registre sont décalés d'un cran dans cette direction. Le bit sortant du registre par la droite va, comme d'habitude, dans le bit de report. A ce moment, le bit qui entre dans le registre par la gauche est la valeur initiale du bit de report (avant qu'il ne soit remplacé par le bit sortant). En mathématique, cela s'appelle une rotation sur neuf bits, puisque les huits bits du registre, plus le neuvième bit (le bit de report), subissent une rotation d'un bit vers la droite. La rotation vers la gauche accomplit la même opération, dans la direction opposée.

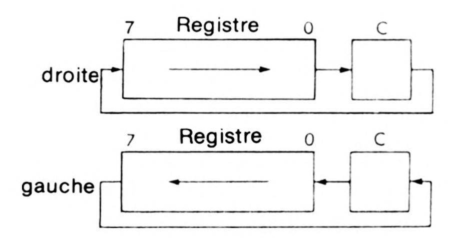


Figure 4.11. — Rotation sur neuf bits

La rotation sur huit bits fonctionne de façon similaire. Le bit 0 est copié dans le bit 7, ou inversement, selon la direction de la rotation. De plus, le bit sortant du registre est également copié dans le bit de report. (voir figure 4.12).

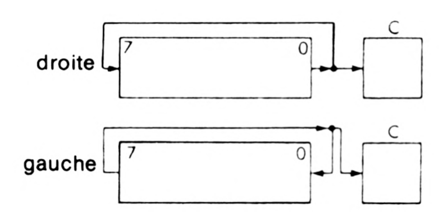


Figure 4.12. — Rotation sur huit bits

Instructions spéciales sur les chiffres

Deux instructions spéciales de rotation de chiffre sont destinées à faciliter l'arithmétique DCB. Elles se traduisent par une rotation de quatre bits, entre les deux chiffres contenus dans l'emplacement mémoire pointé par le registre HL, et un chiffre situé dans la moitié basse de l'accumulateur. (cf. figure 4.13).

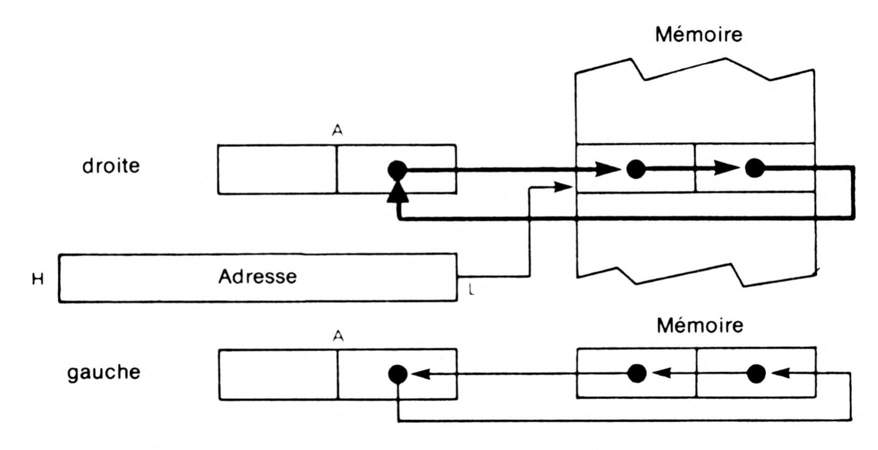


Figure 4.13. — Instructions de rotation de chiffres décimaux

Manipulation de bit

Nous venons de voir la manière dont les opérations logiques peuvent être utilisées pour positionner et effacer des bits, ou des groupes de bits, dans certains registres. Cependant, il est pratique de positionner, ou d'effacer, n'importe quel bit dans n'importe quel registre ou emplacement mémoire, en une seule instruction. Cette opération nécessite un nombre important de codes opératoires, et n'est donc, en général, pas disponible sur les microprocesseurs. Cependant, le Z80 dispose d'importantes capacités de manipulation de bits. [cf. figure 4.14. Ce tableau comprend également les instructions de test qui ne seront décrites qu'au prochain paragraphe].

Deux instructions spéciales de manipulation du bit de report C sont également disponibles : l'inversion (CCF) et le positionnement (SCF) du bit de report (voir figure 4.15).

Ajustement décimal de l'Acc., 'DAA'	27
Complément à un de l'Acc., 'CPL'	2F
Opposé de l'Acc., 'NEG' (complément à 2)	ED 44
Inversion du bit de report, 'CCF'	3F
Positionnement du bit de report, 'SCF'	37

Figure 4.15. — Opérations d'usage général sur AF

	BIT	Registre A	B	C	D	E	H	L	Registre Indirect (HL)	Indexé (IX+d)	(IY+d)
Test 'BIT'	0	CB 47	CB 40	CB 41	CB 42	CB 43	CB 44	CB 45	CB 46	DD CB d 46	FD CB d 46
	1	CB 4F	CB 48	CB 49	CB 4A	CB 4B	CB 4C	CB 4D	CB 4E	DD CB d 4E	FD CB d 4E
	2	CB 57	CB 50	CB 51	CB 52	CB 53	CB 54	CB 55	CB 56	DD CB d 56	FD CB d 56
	3	CB 5F	CB 58	CB 59	CB 5A	CB 5B	CB 5C	CB 5D	CB 5E	DD CB d 5E	FD CB d 5E
	4	CB 67	CB 60	CB 61	CB 62	CB 63	CB 64	CB 65	CB 66	DD CB d 66	FD CB d 66
	5	CB 6F	CB 68	CB 69	CB 6A	CB 6B	CB 6C	CB 6D	CB 6E	DD CB d 6E	FD CB d 6E
	6	CB 77	CB 70	CB 71	CB 72	CB 73	CB 74	CB 75	CB 76	DD CB d 76	FD CB d 76
	7	CB 7F	CB 78	CB 79	CB 7A	CB 7B	CB 7C	CB 7D	CB 7E	DD CB d 7E	FD CB d 7E
Effacement 'RES'	0	CB 87	CB 80	CB 81	CB 82	CB 83	CB 84	CB 85	CB 86	DD CB d 86	FD CB d 86
	1	CB 8F	CB 88	CB 89	CB 8A	CB 8B	CB 8C	CB 8D	CB 8E	DD CB d 8E	FD CB d 8E
	2	CB 97	CB 90	CB 91	CB 92	CB 93	CB 94	CB 95	CB 96	DD CB d 96	FD CB d 96
	3	CB 9F	CB 98	CB 99	CB 9A	CB 9B	CB 9C	CB 9D	CB 9E	DD CB d 9E	FD CB d 9E
	4	CB A7	CB A0	CB A1	CB A2	CB A3	CB A4	CB A5	CB A6	DD CB d A6	FD CB d A6
	5	CB AF	CB A8	CB A9	CB AA	CB AB	CB AC	CB AD	CB AE	DD CB d AE	FD CB d AE
	6	CB B7	CB B0	CB B1	CB B2	CB B3	CB B4	CB B5	CB B6	DD CB d B6	FD CB d B6
	7	CB BF	CB B8	CB B9	CB BA	CB BB	CB BC	CB BD	CB BE	DD CB d BE	FD CB d BE
Positionnement 'SET'	0	CB C7	CB C0	CB C1	CB C2	CB C3	CB C4	CB C5	CB C6	DD CB d C6	FD CB d C6
	1	CB CF	CB C8	CB C9	CB CA	CB CB	CB CC	CB CD	CB CE	DD CB d CE	FD CB d CE
	2	CB D7	CB D0	CB D1	CB D2	CB D3	CB D4	CB D5	CB D6	DD CB d D6	FD CB d D6
	3	CB DF	CB D8	CB D9	CB DA	CB DB	CB DC	CB DD	CB DE	DD CB d DE	FD CB d DE
	4	CB E7	CB E0	CB E1	CB E2	CB E3	CB E4	CB E5	CB E6	DD CB d E6	FD CB d E6
	5	CB EF	CB E8	CB E9	CB EA	CB EB	CB EC	CB ED	CB EE	DD CB d EE	FD CB d EE
	6	CB F7	CB F0	CB F1	CB F2	CB F3	CB F4	CB F5	CB F6	DD CB d F6	FD CB d F6
	7	CB FF	CB F8	CB F9	CB FA	CB FB	CB FC	CB FD	CB FE	DD CB d FE	FD CB d FE

Figure 4.14. — Groupe de manipulation de bit

Test et branchement

Comme les opérations de test font largement appel au registre d'indicateurs, nous décrirons d'abord, en détail, le rôle de chacun de ces derniers. Le contenu du registre d'indicateurs apparaît à la figure 4.16.

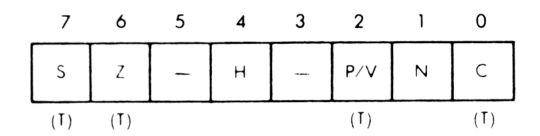


Figure 4.16. — Les registres d'indicateurs

C désigne le report, N l'addition ou la soustraction, P/V la parité ou le débordement, H le report de quartet, Z le zéro, et S le signe. Les bits 3 et 5 du registre d'indicateurs ne sont pas utilisés (« 0 »). Les deux indicateurs H et N sont utilisés par l'arithmétique DCB, et ne peuvent pas être testés. Les quatre autres indicateurs (C, P/V, Z, S) peuvent l'être, lors d'instructions d'appel ou de branchement conditionnels.

Décrivons maintenant le rôle de chaque indicateur.

Report ou retenue (C)

Dans pratiquement tous les microprocesseurs, et dans le Z80 en particulier, le bit de report joue un double rôle. D'une part, il indique qu'une opération d'addition, ou de soustraction, a engendré un report. D'autre part, il sert de neuvième bit, lors des opérations de décalage ou de rotation. Un seul bit, donc, pour deux rôles : certaines opérations, comme la multiplication, s'en trouvent facilitées. L'explication de la multiplication présentée au chapitre précédent en apporte d'ailleurs la confirmation sans équivoque.

Lorsqu'on apprend à utiliser le bit de report, il est important de se souvenir que toutes les opérations arithmétiques le positionnent ou l'effacent, selon le résultat de l'opération. De même que toutes les opérations de décalage, ou de rotation, positionnent ou effacent le bit de report, selon la valeur du bit sortant du registre.

Avec les instructions logiques (AND, OR, XOR), le bit de report est toujours effacé. Elles peuvent donc être utilisées pour effacer explicitement le report.

Les instructions qui affectent le bit de report sont les suivantes : ADD A, s ; ADC A, s ; SUB s ; SBC A, s ; CP s ; NEG ; AND s ; OR s ; XOR s ; ADD DD, ss ; ADC HL, ss ; SBC HL, ss ; RLA ; RLCA ; RRA ; RRCA ; RL m ; RLC m ; RR m ; RRC m ; SLA m ; SRA m ; SRL m ; DDA ; SCF ; CCF.

Soustraction (N)

Cet indicateur n'est normalement pas utilisé par le programmeur. Il l'est de manière interne, par le Z80 lui-même, lors d'opérations DCB. Nous avons vu au chapitre précédent qu'il convient après une addition ou une soustraction DCB, d'exécuter une instruction DAA (Ajustement décimal de l'accumulateur) pour obtenir le résultat DCB correct.

Cependant, cette opération d'« ajustement » n'est pas la même après une addition et après une soustraction. L'instruction DAA est exécutée différemment selon la valeur de l'indicateur N. Ce dernier est mis à « 0 » après une addition, et à « 1 » après une soustraction.

Le symbole utilisé pour cet indicateur [« N »] peut être source de confusion pour les programmeurs habitués à d'autres processeurs, dans la mesure où il peut être confondu avec l'indicateur de signe. Il s'agit, en fait, d'un indicateur de signe à usage interne.

N est mis à « 0 » par : ADD A, s, ADC A, s ; OR s ; XOR s ; INC m ; ADD DD, ss ; ADC HL, ss ; RLA ; RLCA ; RRA ; RRCA ; RL m ; RLC m ; RR m ; RRC m ; SLA m ; SRA m ; SRL m ; RLD ; RRD ; SCF : CCF : IN r, (C) ; LDI ; LDD ; LDIR ; LDDR ; LD A, I ; LD A, R ; BIT b, m.

N est mis à « 1 » par : SUB s ; SBC A, s ; CP s ; AND s ; NEG ; DEC m ; SBC HL, ss ; CPL ; INI ; IND ; OUTI ; OUTD ; INIR ; INDR ; OTIR ; OTDR ; CPI ; CPIR ; CPD ; CPDR.

Parité/débordement (P/V)

L'indicateur de parité/débordement remplit deux fonctions distinctes. Certaines instructions positionnent ou effacent ce bit, selon la parité du résultat ; la parité est déterminée en comptant le nombre de « 1 » contenus dans le résultat. Si ce nombre est impair, l'indicateur de parité est mis à « 0 » (parité impaire). S'il est pair, l'indicateur est mis à « 1 » (parité paire). L'utilisation la plus fréquente de la parité concerne les blocs de caractères (généralement en format ASCII). Le bit de parité est ajouté au code [sur 7 bits] qui représente le caractère de façon à vérifier l'intégrité des données rangées dans un dispositif de mémoire. Par exemple, si un bit du code représentant un caractère est accidentellement changé, à la suite du mauvais fonctionnement du dispositif de mémoire (disque ou une mémoire RAM), ou lors de la transmission, alors le nombre total de « 1 » dans le code de 7 bits est, lui aussi, changé. En contrôlant le bit de parité, l'incohérence sera détectée, et l'erreur signalée. Cet indicateur est notamment utilisé par les instructions logiques et de rotation. Il l'est aussi, bien entendu, lors d'une opération de lecture, à partir d'un dispositif d'entrée-sortie. A la suite de quoi il indique la parité de la donnée qui vient d'être lue.

Remarquons, à l'intention du lecteur habitué au 8080 d'Intel, que l'indicateur de parité du 8080 est utilisé exclusivement en tant que tel, alors que les fonctions de l'indicateur du Z80 sont multiples. Le passage de l'un à l'autre exige donc un certain nombre de précautions.

Dans le cas du Z80, le second rôle essentiel de l'indicateur est de signaler un débordement (qui n'existe pas sur le 8080). L'indicateur de débordement

a été décrit au chapitre 1, en même temps que la notation en complément à deux. Il détecte, lors d'une opération d'addition ou de soustraction, le fait que le signe du résultat soit « accidentellement » changé, à cause d'un débordement du résultat dans le bit de signe. (Souvenez-vous qu'avec une représentation sur huit bits, le plus grand entier positif est + 127, et le plus petit entier négatif − 128, en complément à deux.)

L'indicateur sert aussi, dans le cas du Z80, à deux autres fonctions sans aucun lien avec les précédentes.

Lors de transferts de blocs (LDD, LDDR, LDI, LDIR) ou de recherches par bloc (CPD, CPDR, CPI, CPIR), il détecte, en effet, que le registre qui sert de compteur a atteint la valeur « 0 ». Avec les instructions d'auto-décrément, cet indicateur sera mis à « 0 », si le registre double servant de compteur d'octets contient « 0 ». Avec celles d'auto-incrément, l'indicateur sera positionné si BC − 1 = 0 au début de l'instruction, c'est-à-dire si BC va être décrémenté à « 0 » par l'instruction.

Enfin, lors de l'exécution des deux instructions spéciales LD A, I et LD A, R, l'indicateur P/V reflète la valeur de la bascule d'autorisation des interruptions (IFF2). Ce mécanisme peut être utilisé pour tester, ou sauvegarder, cette valeur.

L'indicateur P est touché par : AND s ; OR s ; XOR s ; RL m ; RLC m ; RR m ; RRC m ; SLA m ; SRA m ; SRL m ; RLD ; RRD ; DAA ; IN r, (C).

L'indicateur V est touché par : ADD A, s ; ADC A, s ; SUB s ; SBC a, s ; CP s ; NEG ; INC m ; DEC m ; ADC HL, ss ; SBC HL, ss.

Il est également utilisé par : LDIR ; LDDR (mise à « 0 ») ; LDI ; LDD ; CPI ; CPIR ; CPD ; CPDR.

Le demi-report (H)

L'indicateur de demi-report indique un éventuel report du bit 3 vers le bit 4, lors d'une opération arithmétique. En d'autres termes, il représente le report du quartet (ou groupe de 4 bits) de poids faible dans celui de poids fort. De toute évidence, son utilisation essentielle est liée aux opérations DCB. Il est notamment utilisé, de façon interne au microprocesseur, par l'instruction d'ajustement décimal de l'accumulateur (DAA), de telle sorte que le résultat soit ajusté à la bonne valeur.

Cet indicateur est positionné par une addition, dans le cas d'un report du bit 3 vers le bit 4, et effacé lorsque ce report n'a pas lieu ; de même, lors d'une soustraction. Cet indicateur est affecté par l'addition, la soustraction, l'incrémentation, la décrémentation, les comparaisons et les opérations logiques.

Les instructions touchant l'indicateur H sont les suivantes : ADD A, s ; SUB s ; SBC A, s ; CP s ; NEG ; AND s ; OR s ; XOR s ; INC m ; DEC m ; RLA ; RLCA ; RRA ; RRCA ; RL m ; RLC m ; RR m ; RRC m ; SLA m ; SR m ; SRL m ; RLD ; RRD ; DAA ; CPL ; SCF ; IN r, (C) ; LDI ; LDD ; LDIR ; LDDR ; LD A, I ; LD A, R ; BIT b, m.

Remarquons que l'indicateur H n'est pas touché par les instructions d'addition et de soustraction sur 16 bits.

Zéro (Z)

L'indicateur Z signale que l'octet qui vient d'être calculé ou transféré a la valeur zéro. Il sert aussi, lors des instructions de comparaison, à indiquer l'égalité et à diverses autres fonctions.

L'indicateur Z est mis à « 1 » chaque fois que l'octet résultat d'une opération ou objet d'un transfert de données est nul. Sinon, il est effacé.

Dans le cas des instructions de comparaison, l'indicateur Z est mis à « 1 » chaque fois que le test indique l'égalité. Sinon, il est mis à « 0 ».

Il intervient, en outre, dans trois autres fonctions. Il est utilisé par l'instruction BIT pour indiquer la valeur du bit qui vient d'être testé. Il est mis à « 1 » si le bit indiqué est « 0 », et effacé sinon.

Avec les instructions spéciales « d'entrée-sortie par blocs » (INI, IND, OUTI, OUTD), l'indicateur Z est positionné si B − 1 = 0, et effacé sinon ; il est positionné lorsque le compteur d'octets est décrémenté à « 0 » (INIR, INDR, OTIR, OTDR).

Enfin, avec l'instruction spéciale IN r, (C), l'indicateur Z est mis à « 1 » pour indiquer que l'octet lu avait la valeur « 0 ».

En résumé, les instructions suivantes conditionnent la valeur de l'indicateur Z : ADD A, s ; ADC A, s ; SUB s ; SBC A, s ; CP s ; NEG ; AND s ; OR s ; XOR s ; INC m ; DEC m ; ADC HL, ss ; SBC HL, ss ; RL m ; RLC m ; RR m ; RRC m ; SLA m ; SRA m ; SRL m ; RLD ; RRD ; DAA ; IN r, (C) ; INI ; IND ; OUTI ; OUTD ; INIR ; INDR ; OTIR ; OTDR ; CPI ; CPIR ; CPD ; CPDR ; LD A, I ; LD A, R ; BIT b, s.

Les instructions usuelles qui restent sans effet sur l'indicateur Z sont : ADD dd, ss ; RLA ; RLCA ; RRA ; CPL ; SCF ; CCF ; LDI ; LDIR ; LDD ; LDDR ; INC dd ; DEC dd.

Signe (S)

Cet indicateur reflète la valeur du bit le plus significatif de l'octet qui vient d'être calculé, ou transféré (bit sept). Dans la notation en complément à deux, le bit le plus significatif sert à représenter le signe. « 0 » indique un nombre positif, et « 1 » un nombre négatif. Pour cette raison, le bit sept est appelé bit de signe.

Lors des communications avec les organes d'entrée-sortie, le bit de signe joue, dans la plupart des microprocesseurs, un rôle important. En effet, ces derniers ne disposent généralement pas d'une instruction BIT pour tester le contenu d'un bit quelconque d'un registre ou de la mémoire. Le bit de signe est donc souvent le plus commode à tester. Lors de l'examen de l'état d'un organe d'entrée-sortie, la lecture du registre d'état positionne automatiquement l'indicateur de signe, qui est mis à la valeur du bit sept du registre d'état. Le programme peut alors le tester facilement. Pour cette raison, dans la plupart des circuits d'entrée-sortie connectés à des systèmes à base de microprocesseurs, l'indicateur le plus important du registre d'état (habituellement l'indicateur prêt/pas prêt) occupe la position du bit sept.

Le Z80 dispose d'une instruction spéciale BIT. Cependant, pour tester un emplacement mémoire (qui pourra être l'adresse du registre d'état d'un

organe d'entrée-sortie), son adresse doit au préalable être chargée dans l'un des registres IX, IY ou HL. Il n'existe pas d'instruction pour tester directement une adresse-mémoire donnée (autrement dit pas de mode d'adressage direct pour cette instruction). L'intérêt qu'il y a à placer l'indicateur prêt d'un organe d'entrée-sortie à la position du bit sept demeure. Même dans le cas du Z80.

L'indicateur de signe est, enfin, utilisé par l'instruction spéciale IN r, (C) pour indiquer le signe de la donnée lue.

Les instructions qui touchent l'indicateur de signe sont : ADD A, s ; SUB s ; SBC A, s ; CP s ; NEG ; AND s ; OR s ; XOR s ; INC m ; DEC m ; ADC HL, ss ; SBC HL ss ; RL m ; RLC m ; RR m ; RRC m ; SLA m ; SRA m ; SRL m ; RLD ; RRD ; DDA ; IN r, (C) ; CPR ; CPIR ; CPD ; CPDR ; LD A, I ; LD A, R.

Résumé des indicateurs

Les indicateurs sont utilisés pour détecter automatiquement des conditions spéciales de l'unité arithmétique et logique du microprocesseur. Ils peuvent être commodément testés par des instructions spécialisées. Des actions spécifiques peuvent ainsi être entreprises pour tenir compte des conditions détectées. Il est important de comprendre le rôle des divers indicateurs disponibles, dans la mesure où la plupart des décisions prises par le programme le seront en fonction d'eux. La seule exception concerne le mécanisme d'interruption, qui sera décrit dans le chapitre consacré aux entrées-sorties. Ce mécanisme est susceptible d'engendrer des branchements à des adresses spécifiques, chaque fois qu'un signal physique est reçu sur les broches spécialisées du Z80.

Pour l'instant, il suffit de garder à l'esprit la fonction principale de chaque indicateur. Au cours de la programmation le lecteur pourra se reporter, à la description de ces instructions, dans la deuxième partie de ce chapitre. Il pourra ainsi vérifier leur effet sur les divers indicateurs. La plupart du temps, il est possible de ne pas tenir compte de la plupart des indicateurs. Le lecteur peu familiarisé avec eux n'a aucune raison de se laisser impressionner par leur apparente complexité. Leurs conditions d'utilisation se clarifieront au fur et à mesure que nous examinerons des programmes d'application.

La figure 4.17 donne un récapitulatif des six indicateurs, et de la façon dont ils sont positionnés, ou effacés, par les diverses instructions.

Les instructions de saut

Une instruction de branchement produit un branchement forcé à une adresse indiquée du programme. Elle change le cours normal de son exécution. Le programme passe d'un mode séquentiel à un mode où, soudainement, un segment de programme différent est exécuté. Les sauts peuvent être conditionnels ou inconditionnels. Dans la deuxième hypothèse, le branchement à une adresse spécifique est opéré sans tenir compte d'aucune autre condition.

INSTRUCTION	C	Z	P/V	S	N	H	COMMENTAIRES
ADD A, s, ADC A, s	↕	↕	V	↕	0	↕	Addition sur 8 bits incluant ou non le report
SUB s, SBC A, s, CP s, NEG	↕	↕	V	↕	1	↕	Soustraction sur 8 bits incluant ou non le report, comparaison, opposé de l'accumulateur
AND s	0	↕	P	↕	0	1	Opérations logiques et positionnement de différents indicateurs
OR s, XOR s	0	↕	P	↕	0	0	
INC s	•	↕	V	↕	0	↕	Incrémentation sur 8 bits
DEC m	•	↕	V	↕	1	↕	Décrémentation sur 8 bits
ADD DD, ss	↕	•	•	•	0	X	Addition sur 16 bits
ADC HL, ss	↕	↕	V	↕	0	X	Addition sur 16 bits incluant le report
SBC HL, ss	↕	↕	V	↕	1	X	
RLA, RLCA, RRA, RRCA	↕	•	•	•	0	0	Rotation de l'accumulateur
RL m, RLC m, RR m, RRC m SLA m, SRA m, SRL m	↕	↕	P	↕	0	0	Rotation et décalage de l'emplacement m
RLD, RRD	•	↕	P	↕	0	0	Rotation à gauche ou à droite de chiffres
DAA	↕	↕	P	↕	•	↕	Ajustement décimal de l'accumulateur
CPL	•	•	•	•	1	1	
SCF	1	•	•	•	0	0	Positionnement du report
CCF	↕	•	•	•	0	X	Complémenter le report
IN r, (C)	•	↕	P	↕	0	0	Entrée indirecte par registre
INI, IND, OUTI, OUTD	•	↕	X	X	1	X	Entrée sortie par bloc
INIR, INDR, OTIR, OTDR	•	1	X	X	1	X	Z = 0 si B ≠ 0, sinon Z = 1
LDI, LDD	•	X	↕	X	0	0	Transfert de bloc
LDIR, LDDR	•	X	0	X	0	0	P/V = 1 si BC ≠ 0 sinon P/V = 0
CPI, CPIR, CPD, CPDR	•	↕	↕	↕	1	X	Instruction de recherche dans un bloc Z = 1 si A = (HL) sinon Z = 0 P/V = 1 si BC ≠ 0 sinon PV = 0
LD A, I, LD A, R	•	↕	IFF	↕	0	0	Le contenu de la bascule d'autorisation des interruptions est copié dans l'indicateur P/V
BIT b, s	•	↕	X	X	0	1	L'inverse du bit b de l'emplacement m est copié dans l'indicateur Z

Les notations suivantes sont utilisées dans cette table :

SYMBOLE	SIGNIFICATION
C	Indicateur de Report C = 1 si l'opération génère un report du bit le plus significatif de l'opérande ou du résultat.
Z	Indicateur de Zéro Z = 1 si le résultat de l'opération est zéro.
S	Indicateur de signe S = 1 si le bit le plus significatif du résultat est zéro.
P/V	Indicateur de parité ou de débordement. La parité (P) et le débordement (V) partagent le même indicateur. Les opérations logiques affectent cet indicateur selon la parité du résultat. Alors que les opérations arithmétiques l'affectent en fonction du débordement du résultat. Si P/V reflète la parité, P/V = 1 si la parité du résultat de l'opération est paire, P/V = 0 si elle est impaire ; Si P/V reflète le débordement, P/V = 1 si le résultat de l'opération provoque un débordement.
H	Indicateur de demi-report H = 1 si l'opération d'addition ou de soustraction génère un report du bit 3 de l'accumulateur.
N	Indicateur d'addition/soustraction. N = 1 si l'opération précédente était une soustraction. Les indicateurs H et N sont utilisés, conjointement à l'instruction d'ajustement décimal de l'accumulateur, pour corriger le résultat en format DCB compacté correct après une addition ou une soustraction sur des opérandes en format DCB compacté.
↕	L'indicateur est affecté selon le résultat de l'opération.
•	L'indicateur n'est pas changé par l'opération.
0	L'indicateur est effacé par l'opération.
1	L'indicateur est positionné par l'opération.
X	L'indicateur n'a pas d'importance.
V	L'indicateur P/V est affecté selon le débordement du résultat de l'opération.
P	L'indicateur P/V est affecté selon la parité du résultat de l'opération.
r	L'un quelconque des registres A, B, C, D, E, H, L, de l'unité centrale.
s	Un emplacement quelconque de 8 bits pour tous les modes d'adressage autorisés pour cette instruction.
ss	Un emplacement quelconque de 16 bits pour tous les modes d'adressage autorisés pour cette instruction.
ii	L'un quelconque des deux registres d'index IX ou IY.
R	Compteur de rafraîchissement.
n	Valeur sur 8 bits dans l'intervalle (0-255).
nn	Valeur sur 16 bits dans l'intervalle (0-65535).
m	Un emplacement quelconque de 8 bits pour tous les modes d'adressage autorisés pour cette instruction.

Figure 4.17. — Résumé du comportement des indicateurs

Dans un saut conditionnel, le branchement à une adresse spécifique n'a lieu que si une ou plusieurs conditions sont vérifiées. C'est le type d'instruction de saut utilisée pour prendre des décisions en fonction de la donnée traitée ou du résultat obtenu.

Pour expliquer les instructions de saut conditionnel, il est nécessaire de comprendre le rôle du registre d'indicateurs, dans la mesure où toutes les décisions de branchement reposent sur ces derniers. Tel était l'objectif du paragraphe précédent. Nous pouvons maintenant examiner, plus en détail, les instructions de saut disponibles sur le Z80.

Il convient de distinguer deux types principaux : les instructions de saut à l'intérieur du programme principal (que l'on appelle « sauts »), et le type spécial d'instructions utilisé pour se brancher à un sous-programme et en revenir (« appel » et « retour »). Toutes les instructions de saut ont le même effet : le compteur ordinal PC est chargé avec une nouvelle adresse, à partir de laquelle l'exécution normale du programme se poursuit. La puissance véritable des diverses instructions de saut ne peut être bien appréciée que dans le contexte des différents modes d'adressage fournis par le microprocesseur. Nous différerons l'examen de cet aspect de la question jusqu'au prochain chapitre, où il sera question des modes d'adressage. Nous ne prendrons en compte, ici, que les autres aspects.

Les sauts peuvent être inconditionnels (branchement à une adresse mémoire spécifiée) ou conditionnels. Dans le second cas, il est possible de tester l'un des quatre indicateurs [Z, C, P/V et S], pour déterminer si sa valeur est « 0 » ou « 1 ».

Les abréviations correspondantes sont les suivantes :

Z = nul (Z = 1)
NZ = non nul (Z = 0)
C = Report (C = 1)
NC = pas de report (C = 0)
PO = Parité paire
PE = Parité impaire
P = Positif (S = 0)
M = Négatif (S = 1)

Le Z80 dispose, en outre, d'une instruction spéciale qui décrémente le registre B, et effectue un saut à une adresse spécifiée s'il n'atteint pas la valeur zéro. Il s'agit d'une instruction puissante, utilisée pour terminer une boucle ; nous y avons déjà eu recours plusieurs fois, dans le chapitre précédent : c'est l'instruction DJNZ.

De même, les instructions CALL (Appel) et RET (retour) peuvent être conditionnelles ou inconditionnelles. Elles testent les mêmes indicateurs que les instructions de branchement.

Dans un ordinateur, l'existence de branchements conditionnels est une propriété importante, qui n'existe généralement pas sur les autres microprocesseurs. Elle augmente l'efficacité des programmes, en réalisant en une seule instruction ce qui en nécessite habituellement deux.

Il existe enfin, dans le cas des routines d'interruption, deux instructions spéciales de retour : RETI et RETN. Elles seront décrites dans le paragraphe du chapitre 6 consacré aux interruptions.

La figure 4.18 montre les modes d'adressage et les codes opératoires des divers branchements.

			Condition									
			inconditionnel	report	pas de report	Nul	Non Nul	Parité paire	Parité impaire	Signe négatif	Signe positif	Registre B ≠ 0
Saut 'JP'	immédiat étendu	nn	C3 n n	DA n n	D2 n n	CA n n	C2 n n	EA n n	E2 n n	FA n n	F2 n n	
Saut 'JR'	relatif	PC+e	18 e-2	38 e-2	30 e-2	28 e-2	20 e-2					
Saut 'JP'	Registre indirect	(HL)	E9									
Saut 'JP'	Registre indirect	(IX)	DD E9									
Saut 'JP'	Registre indirect	(IY)	FD E9									
'CALL'	immédiat étendu	nn	CD n n	DC n n	D4 n n	CC n n	C4 n n	EC n n	E4 n n	FC n n	F4 n n	
Documentation de B, saut si non nul 'DJNZ'	Relatif	PC+e										10 e-2
Retour 'RET'	Registre indirect	(SP) (SP+1)	C9	D8	D0	C8	C0	E8	E0	F8	F0	
Retour d'interruption 'RETI'	Registre indirect	(SP) (SP+1)	ED 4D									
Retour d'interruption non masquable 'RETN'	Registre indirect	(SP) (SP+1)	ED 45									

Figure 4.18. — Instructions de saut

La discussion détaillée des divers modes d'adressage fera l'objet du chapitre 5.

En examinant la figure 4.18, il apparaît que de nombreux modes d'adressage comportent des restrictions. Par exemple, le saut absolu JP nn est capable de tester quatre indicateurs, alors que JR ne peut en tester que deux.

Une remarque importante : il est tentant d'utiliser JR aussi souvent que possible, dans la mesure où il s'agit d'une instruction plus courte que JP (un octet de moins). Le relogement du programme en est facilité. Cependant, JR et JP ne sont pas interchangeables : JR est ainsi incapable de tester les indicateurs de parité ou de signe.

Autre type de branchement spécialisé : l'instruction RST (en anglais restart ou « repartir »). Elle ne requiert qu'un seul octet, et permet de se brancher à l'une quelconque des huit adresses de départ, à l'extrémité basse de la mémoire. Ces adresses de départ sont, en décimal, 0, 8, 16, 24, 32, 40,

48 et 56. Il s'agit d'une instruction puissante, puisqu'elle est exprimée en un seul octet. C'est le branchement le plus rapide qui soit, et pour cette raison, il est principalement utilisé pour répondre aux interruptions. Cependant, le programmeur peut en disposer à d'autres fins. La figure 4.19 en recense les codes opératoires.

adresse de départ	OP CODE	
0000_H	C7	'RST 0'
0008_H	CF	'RST 8'
0010_H	D7	'RST 16'
0018_H	DF	'RST 24'
0020_H	E7	'RST 32'
0028_H	EF	'RST 40'
0030_H	F7	'RST 48'
0038_H	FF	'RST 56'

Figure 4.19. — Groupe des « restart »

Instructions d'entrée-sortie

Les techniques d'entrée-sortie seront décrites, en détail, au chapitre 6. Sommairement, les organes d'entrée-sortie peuvent être adressés de deux façons : comme des emplacements mémoire, au moyen de l'une des instructions déjà décrites, ou en mettant en œuvre des instructions spécifiques d'entrée-sortie. Les instructions usuelles adressant la mémoire utilisent trois octets : un pour le code opératoire, et deux pour l'adresse. Elles nécessitent donc trois accès mémoire. D'où la longueur de leur exécution. Le but essentiel des instructions spécialisées est de fournir des instructions plus courtes, et donc plus rapides. Il reste que les instructions d'E/S présentent deux désavantages.

D'une part, elles « gaspillent » plusieurs des précieux, parce que peu nombreux, codes opératoires disponibles (en général, seuls 8 bits sont

utilisés pour fournir tous les codes opératoires nécessaires à un microprocesseur). D'autre part, elles nécessitent la génération d'un ou de plusieurs signaux spécialisés d'entrée-sortie. De là un nouveau « gaspillage » d'une ou de plusieurs des broches disponibles sur un microprocesseur. Le nombre de broches est généralement limité à 40. En raison de tous ces désavantages possibles, la plupart des microprocesseurs n'offrent pas d'instructions spécifiques d'entrée-sortie. Le 8080 (le premier microprocesseur huit bits d'usage général puissant qui ait été introduit sur le marché) en fournit cependant, ainsi que le Z80, dont nous savons qu'il est compatible avec le premier nommé.

L'avantage des instructions d'entrée-sortie est une exécution plus rapide, puisqu'elles ne requièrent que deux octets. Un résultat identique peut cependant être obtenu avec un mode d'adressage spécial, appelé adressage « en page 0 », dans lequel le champ de l'adresse est limité à huit bits. Cette solution est souvent retenue dans d'autres microprocesseurs.

Les deux instructions d'entrée-sortie de base sont IN et OUT. Elles transfèrent, respectivement, le contenu de l'emplacement d'entrée-sortie désigné dans l'un des registres de travail, et celui du registre, dans l'organe d'entrée-sortie. Elles occupent, bien sûr, deux octets. Le premier est réservé au code opératoire. Le second constitue la partie basse de l'adresse, alors que l'accumulateur en représente la partie haute. Il est ainsi possible de sélectionner un des 64 K organes périphériques. Cela nécessite que l'accumulateur soit chargé, à chaque fois, avec le contenu approprié. L'exécution peut s'en trouver ralentie.

Le Z80 offre, de surcroît, un mode registre indirect, et quatre instructions spécialisées de transfert par bloc, en entrée ou en sortie.

Dans le mode *registre indirect,* dont le format est IN r, (C), le registre double BC sert de pointeur sur l'organe d'entrée-sortie. Le contenu de B est placé sur les parties hautes du bus adresse. Celui du dispositif d'entrée-sortie désigné est alors chargé dans le registre désigné par r.

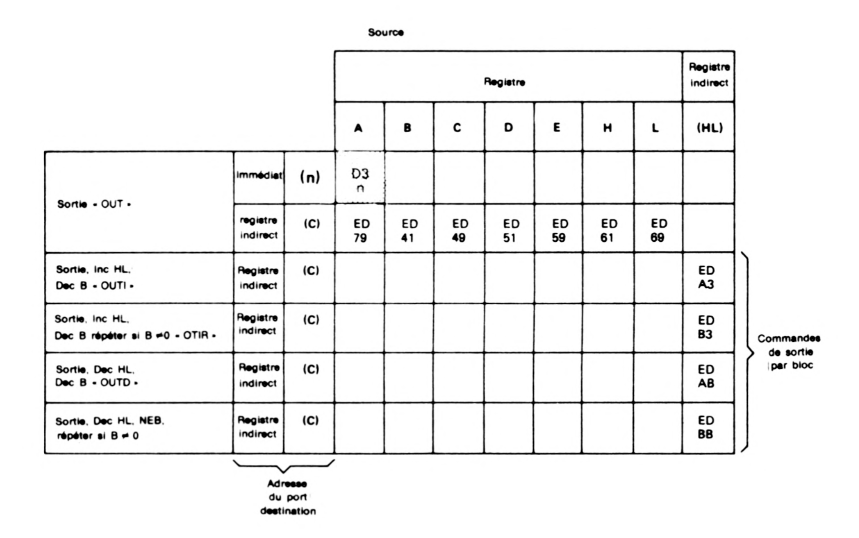
Il en est de même de l'instruction OUT.

Les quatre instructions de transfert par bloc en entrée sont : INI, INIR (INI répété), IND et INDR (IND répété). Il s'agit, de même, en sortie, de : OUTI, OTIR, OUTD, OTDR.

Dans ce mode automatique de transfert par bloc, le registre double HL sert de pointeur sur la destination, et le registre C à sélectionner l'organe d'entrée-sortie (un parmi 256). Dans le cas de l'instruction de sortie, HL pointe sur l'origine. Le registre B sert de compteur, et peut être incrémenté ou décrémenté. Les instructions correspondantes en entrée sont INI (incrémentation), et IND (décrémentation).

INI est une instruction de transfert automatique d'un seul octet. Le registre C opère la sélection de l'organe d'entrée. Un octet est lu à partir de cet organe, et rangé dans l'adresse mémoire pointée par H et L. HL est alors incrémenté de 1, et le compteur B décrémenté de 1.

INIR est la même instruction, automatisée. Elle est exécutée de façon répétitive, jusqu'à ce que le compteur, en se décrémentant, atteigne « 0 ». Il est ainsi possible de transférer automatiquement jusqu'à 256 octets. Notons

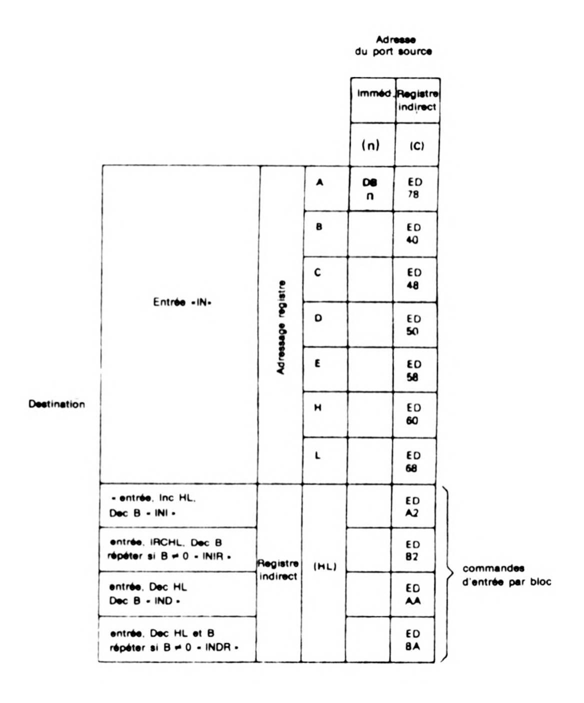


			Source: Registre A	B	C	D	E	H	L	Registre indirect (HL)
Sortie « OUT »	Immédiat	(n)	D3 n							
Sortie « OUT »	registre indirect	(C)	ED 79	ED 41	ED 49	ED 51	ED 59	ED 61	ED 69	
Sortie, Inc HL, Dec B « OUTI »	Registre indirect	(C)								ED A3
Sortie, Inc HL, Dec B répéter si B ≠0 « OTIR »	Registre indirect	(C)								ED B3
Sortie, Dec HL, Dec B « OUTD »	Registre indirect	(C)								ED AB
Sortie, Dec HL, NEB, répéter si B ≠ 0	Registre indirect	(C)								ED BB

Adresse du port destination

Commandes de sortie par bloc (OUTI, OTIR, OUTD, répéter si B ≠ 0)

Figure 4.20. — Groupe des sorties



Adresse du port source

Destination			Imméd. (n)	Registre indirect (C)
Entrée « IN »	Adressage registre	A	DB n	ED 78
Entrée « IN »	Adressage registre	B		ED 40
Entrée « IN »	Adressage registre	C		ED 48
Entrée « IN »	Adressage registre	D		ED 50
Entrée « IN »	Adressage registre	E		ED 58
Entrée « IN »	Adressage registre	H		ED 60
Entrée « IN »	Adressage registre	L		ED 68
« entrée, Inc HL, Dec B « INI »	Registre indirect	(HL)		ED A2
entrée, IRCHL, Dec B répéter si B ≠ 0 « INIR »	Registre indirect	(HL)		ED B2
entrée, Dec HL Dec B « IND »	Registre indirect	(HL)		ED AA
entrée, Dec HL et B répéter si B ≠ 0 « INDR »	Registre indirect	(HL)		ED BA

commandes d'entrée par bloc (INI, INIR, IND, INDR)

Figure 4.21. — Groupe des entrées

que, pour obtenir ce total optimum, le registre B doit être mis à la valeur « 0 », préalablement à l'exécution de l'instruction.

Les figures 4.20 et 4.21 recensent les codes opératoires des instructions d'entrée-sortie.

Les instructions de contrôle

Les instructions de contrôle modifient le mode d'opération de l'unité centrale, dont l'état interne est manipulé comme une information. Ces instructions sont au nombre de sept.

L'instruction NOP est une instruction signifiant : « pas d'opération ». Elle ne fait rien pendant tout un cycle.

Son rôle, de manière caractéristique, consiste, soit à introduire délibérément un délai (4 états = 2 microsecondes avec une horloge à 2 MHz), soit à remplir les espaces créés dans un programme pendant la phase de mise au point. Pour faciliter cette mise au point, le code opératoire du NOP est traditionnellement un ensemble de zéro. Pourquoi ? Parce qu'au moment de l'exécution, la mémoire est souvent effacée : elle ne contient que des zéros. L'exécution de NOP nous assure qu'aucun dommage n'est causé, et que l'exécution du programme n'est pas arrêtée.

L'instruction HALT est utilisée conjointement aux interruptions et à la réinitialisation. Elle suspend réellement le fonctionnement de l'unité centrale. L'UC ne repartira qu'à la réception d'une interruption, ou d'un signal de réinitialisation. Dans ce mode, l'unité centrale exécute continuellement des NOP.

Une instruction HALT est souvent placée à la fin des programmes, pendant la phase de mise au point, dans la mesure où le programme principal n'a habituellement plus rien d'autre à faire. Ce dernier doit alors être explicitement redémarré.

Deux instructions spécialisées positionnent la bascule interne, de façon à autoriser, ou à interdire, les interruptions. Ce sont EI et DI. Les interruptions seront décrites au chapitre 6. La bascule d'interruption autorise ou interdit l'interruption d'un programme. Pour empêcher les interruptions de se produire pendant une partie déterminée d'un programme, la bascule pourra être invalidée par l'instruction DI. Nous y reviendrons, au chapitre 6. La figure 4.22 recense ces instructions.

Le Z80 dispose enfin de trois modes d'interruption (le 8080 n'en dispose que d'un seul). Le mode d'interruption 0 est celui du 8080 ; le mode d'interruption 1 est un appel à l'adresse 038 H, et le mode d'interruption 2 un appel indirect qui utilise le contenu d'un registre spécial I, plus 8 bits fournis par le dispositif interrompant, comme un pointeur indirect à la routine d'interruption en mémoire. Ces modes seront expliqués au chapitre 6.

NOP	00	
HALT	76	
Interdire les interruptions DI	F3	
Autoriser les interruptions EI	FB	
Activer le Mode d'interruption 0 IM	ED 46	Mode du 8080A
Activer le Mode d'interruption 1 'IM1'	ED 56	Appel à l'adresse 0038_H
Activer le Mode d'interruption 2 'IM2'	ED 5E	Appel indirect en utilisant le registre I et les 8 bits du périphérique interrompant comme pointeur

Figure 4.22. — Contrôle divers de l'UC

Des broches spéciales [IRQ et NMI] déclenchent un mécanisme d'interruption qui sera également expliqué au chapitre 6.

RÉSUMÉ

Les cinq catégories d'instructions disponibles dans le Z80 ont maintenant été décrites. Des détails sur chacune d'entre elles seront donnés dans la section suivante. Il n'est pas indispensable de comprendre le rôle de chaque instruction pour commencer un programme. La connaissance de quelques instructions essentielles de chacun des types est suffisante, au départ. Cependant, lorsque vous commencerez à écrire des programmes par vous-même, vous devrez apprendre toutes les instructions du Z80, pour que le résultat de votre travail soit de qualité. Bien évidemment, l'efficacité n'est pas, au départ, importante. C'est pourquoi la plupart des instructions peuvent être ignorées.

Un aspect important n'a pas encore été évoqué. Il s'agit de l'ensemble des techniques d'adressage implantées sur le Z80 pour faciliter l'accès aux données, à l'intérieur de l'espace mémoire. Elles feront l'objet du prochain chapitre.

LES INSTRUCTIONS DU Z80 : DESCRIPTION INDIVIDUELLE

ABRÉVIATIONS

Indicateur	Positionné	Non positionné
Report	C (Report)	NC (Pas de report)
Signe	M (moins)	P (plus)
Zéro	Z (nul)	NZ (non nul)
Parité	PE (paire)	PO (impaire)

● Modifié selon le résultat de l'opération
○ indicateur mis à zéro
1 indicateur mis à un
? indicateur modifié aléatoirement par l'opération
X cas spécial, voir la note d'accompagnement sur la même page.

Les bits 3 et 5 du registre d'indicateurs ont toujours une valeur aléatoire.

ADC A, s Additionner l'accumulateur et l'opérande spécifié avec le report.

Fonction : A ← A + s + C

Format : *s* peut être : r, n, (HL), (IX + d) ou (IY + d)

r	1 0 0 0 1 ← r →	
n	1 1 0 0 1 1 1 0	octet 1 : CE
	← n →	octet 2 : valeur immédiate
(HL)	1 0 0 0 1 1 1 0	8E
(IX + d)	1 1 0 1 1 1 0 1	octet 1 : DD
	1 0 0 0 1 1 1 0	octet 2 : 8E
	← d →	octet 3 : déplacement
(IY + d)	1 1 1 1 1 1 0 1	octet 1 : FD
	1 0 0 0 1 1 1 0	octet 2 : 8E
	← d →	octet 3 : déplacement

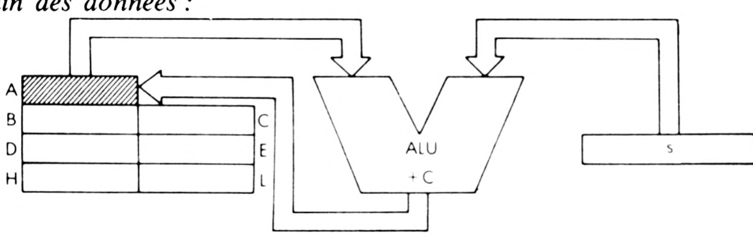
r peut être n'importe lequel de :

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Description : L'opérande s et l'indicateur de report c du registre d'état sont additionnés à l'accumulateur, et le résultat est rangé dans l'accumulateur. s est défini dans la description des instructions similaires ADD.

Chemin des données :





Durée :

s:	cycles M :	temps T :	usec @ 2 MHz:
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Mode d'adressage : r : implicite ; n : immédiat ; (HL) : indirect ; (IX + d), (IY + d) : indexé.

Codes : ADC A,r

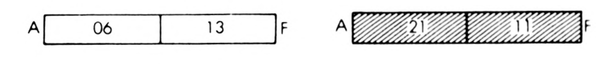
r:	A	B	C	D	E	H	L
	8F	88	89	8A	8B	8C	8D

Indicateurs :

S	Z		H		P/V	N	C
●	●		●		●	○	●

Exemple : ADC A, 1A

Avant : Après :



CE
1A

CODE OBJET

ADC HL, ss Additionner HL et le registre double ss avec le report.

Fonction : HL ← HL + ss + C

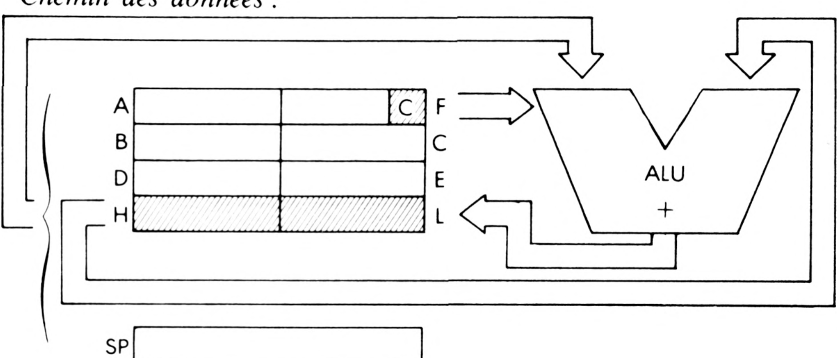
Format :

1	1	1	0	1	1	0	1	octet 1 : ED
0	1	s	s	1	0	1	0	octet 2

Description : Le contenu du registre double HL est additionné au contenu du registre double spécifié, puis la valeur de l'indicateur de report est ajoutée. Le résultat final est rangé à nouveau dans HL. ss peut être n'importe lequel de :

BC – 00 HL – 10
DE – 01 SP – 11

Chemin des données :



Durée : 4 cycles M ; 15 temps T : 75 usec @ 2 MHz

Mode d'adressage : Implicite

Codes : ss:

	BC	DE	HL	SP
ED-	4A	5A	6A	7A

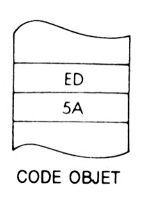
Indicateurs :

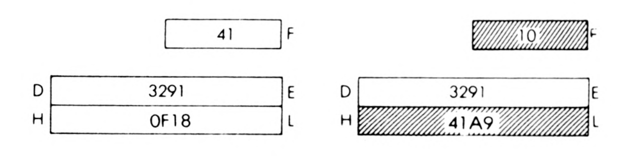
S	Z		H		P/V	N	C
●	●		●		●	○	●

H est positionné s'il y a un report du bit 11

Exemple : ADC HL, DE

Avant : Après :





ADD A, (HL) Additionner l'accumulateur et l'emplacement mémoire d'adresse indirecte (HL).

Fonction : A ← A + (HL)

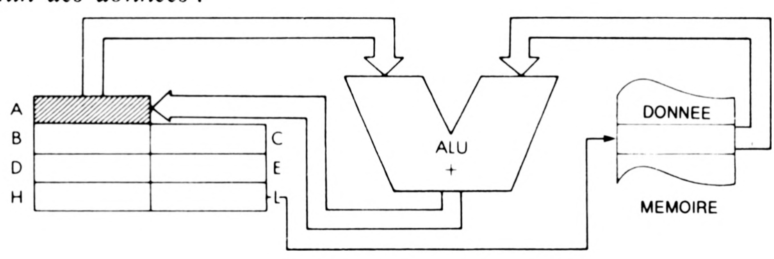
Format :

1	0	0	0	0	1	1	0

86

Description : Le contenu de l'accumulateur est additionné au contenu de l'emplacement mémoire adressé par le registre double HL. Le résultat est rangé dans l'accumulateur.

Chemin des données :



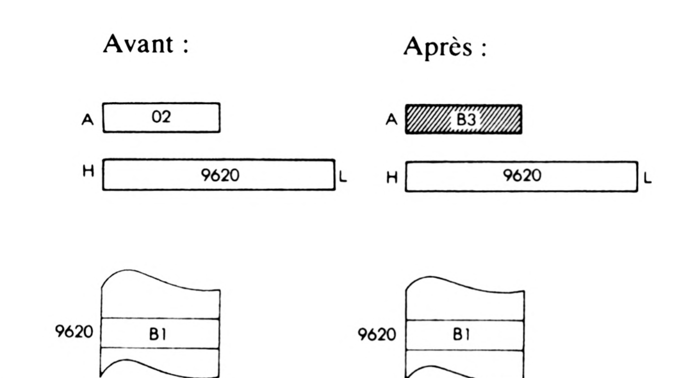
Durée : 2 cycles M ; 7 temps T : 3,5 usec @ 2 MHz

Mode d'adressage : Indirect.

Indicateurs :

S	Z		H		P/V	N	C
●	●		●		●	○	●

Exemple : ADD A, (HL)



86

CODE OBJET

ADD A, (IX + d) Additionner l'accumulateur et l'emplacement mémoire d'adresse indexée (IX + d).

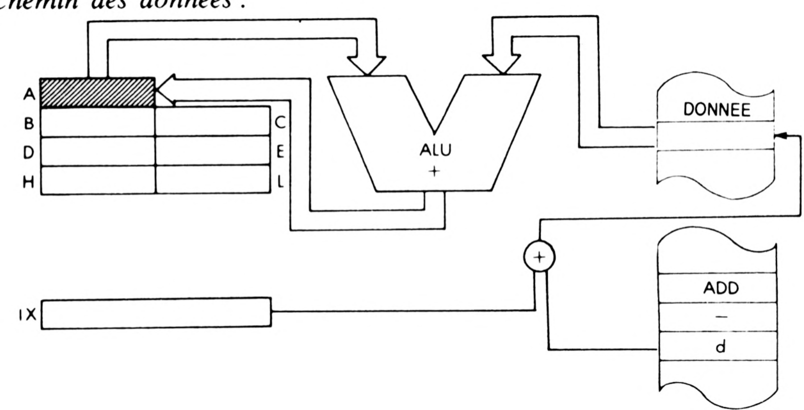
Fonction : A ← A + (IX + d)

Format :

1	1	0	1	1	1	0	1	octet 1 : DD
1	0	0	0	0	1	1	0	octet 2 : 86
←			d				→	octet 3 : déplacement

Description : Le contenu de l'accumulateur est additionné au contenu de l'emplacement mémoire adressé par le contenu du registre IX plus le déplacement immédiat. Le résultat est rangé dans l'accumulateur.

Chemin des données :



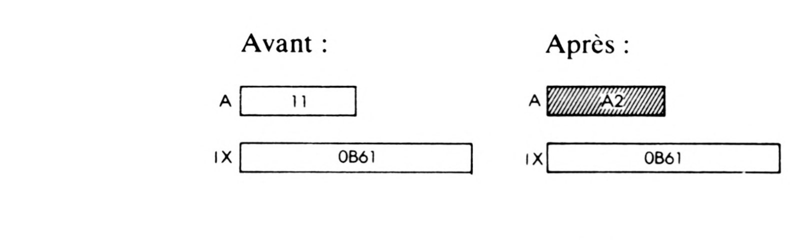
Durée : 5 cycles M ; 19 temps T : 9,5 usec @ 2 MHz

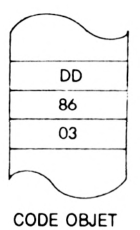
Mode d'adressage : Indexé.

Indicateurs :

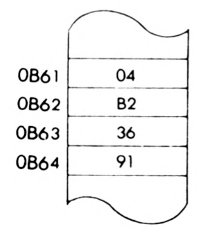
S	Z		H		P/V	N	C
●	●		●		●	○	●

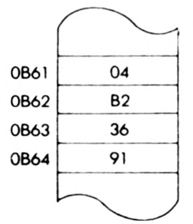
Exemple : ADD A, (IX + 3)





CODE OBJET





ADD A, (IY + d) Additionner l'accumulateur et l'emplacement mémoire d'adresse indexée (IY + d).

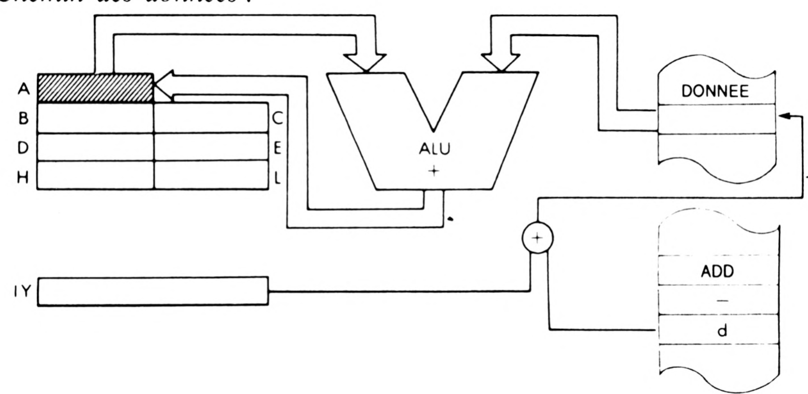
Fonction : A ← A + (IY + d)

Format :

1	1	1	1	1	1	0	1	octet 1 : FD
1	0	0	0	0	1	1	0	octet 2 : 86
← d →								octet 3 : déplacement

Description : Le contenu de l'accumulateur est additionné au contenu de l'emplacement mémoire adressé par le contenu du registre IY plus le déplacement immédiat. Le résultat est rangé dans l'accumulateur.

Chemin des données :



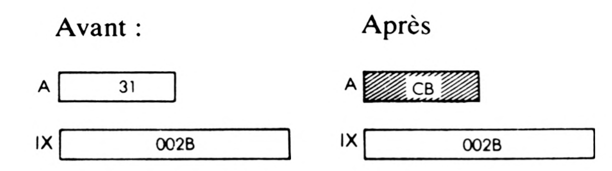
Durée : 5 cycles M ; 19 temps T ; 9,5 usec @ 2 MHz

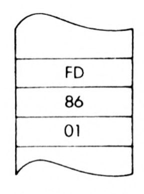
Mode d'adressage : Indexé.

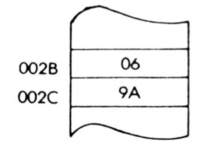
Indicateurs :

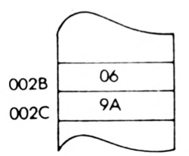
S	Z		H		P/V	N	C
●	●		●		●	○	●

Exemple : ADD A, (IY + 1)





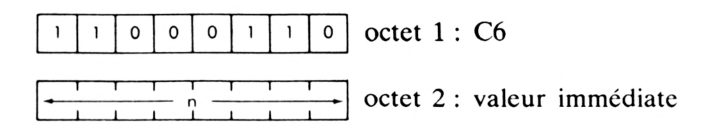




ADD A, n Additionner l'accumulateur et la valeur immédiate n.

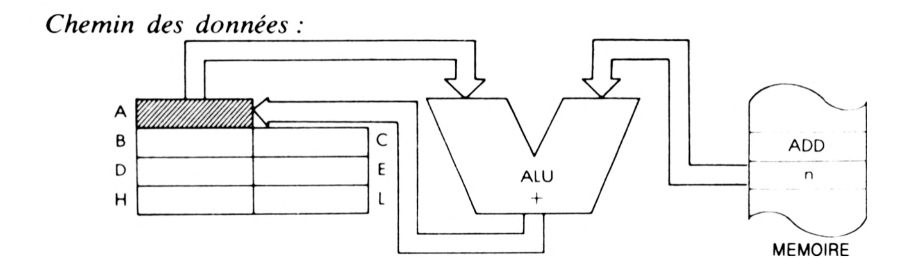
Fonction : A ← A + n

Format :



Description : Le contenu de l'accumulateur est additionné au contenu de l'emplacement mémoire suivant immédiatement le code opération. Le résultat est rangé dans l'accumulateur.

Chemin des données :



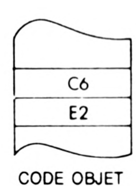
Durée : 2 cycles M ; 7 temps T : 3,5 usec @ 2 MHz

Mode d'adressage : Immediat.

Indicateurs :

S	Z		H		P/(V)	N	C
●	●		●		●	○	●

Exemple : ADD A, E2



Avant : A 43

Après : A 25

ADD A, r

Additionner l'accumulateur et le registre r.

Fonction : A ← A + r

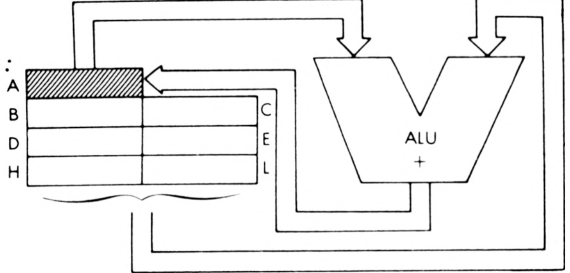
Format :

1	0	0	0	0	← r →

Description : Le contenu de l'accumulateur est additionné au contenu du registre spécifié. Le résultat est placé dans l'accumulateur r peut être n'importe lequel de :

A – 111 E – 011
B – 000 H – 100
C – 001 L – 101
D – 010

Chemin des données :



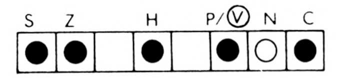
Durée : 1 cycle M ; 4 temps T : 2 usec @ 2 MHz.

Mode d'adressage : Implicite

Codes :

A	B	C	D	E	H	L
87	80	81	82	83	84	85

Indicateurs :



Exemple : ADD A, B

Avant :

Après :

ADD HL, ss Additionner HL et le registre double ss.

Fonction : HL ← HL + ss

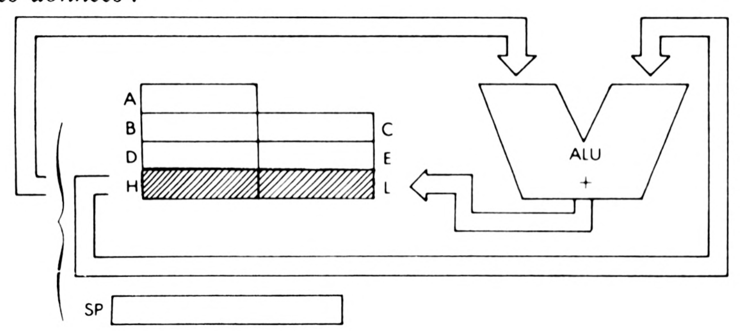
Format :

0	0	S	S	1	0	0	1

Description : Le contenu du registre double spécifié est additionné au contenu du registre double HL et le résultat est rangé dans HL. ss peut être n'importe lequel de :

BC – 00 HL – 10
DE – 01 SP – 11

Chemin des données :



Durée : 3 cycles M ; 11 temps T : 5,5 usec @ 2 MHz

Mode d'adressage : Implicite

Codes : ss:

BC	DE	HL	SP
09	19	29	39

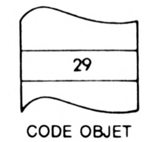
Indicateurs :

S	Z		H		P/V	N	C
			●			○	●

C est positionné par le report du bit 15, effacé autrement.
H est positionné par un report du bit 11

Exemple : ADD HL, HL

Avant : Après :



H 06B1 L

H 0C62 L

ADD IX, rr

Additionner IX et le registre double rr.

Fonction : IX ← IX + rr

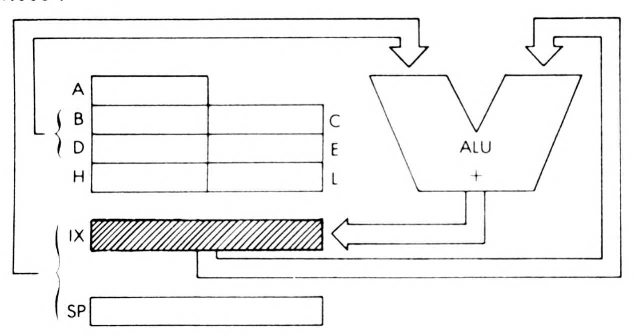
Format :

1	1	0	1	1	1	0	1	octet 1 : DD
0	0	r	r	1	0	0	1	octet 2

Description : Le contenu du registre IX est additionné au contenu du registre double spécifié et le résultat est rangé à nouveau dans IX. rr peut être n'importe lequel de :

BC – 00 IX – 10
DE – 01 SP – 11

Chemin des données :



Durée : 4 cycles M ; 15 temps T : 7,5 usec @ 2 MHz

Mode d'adressage : Implicite

Codes :

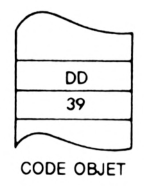
rr:	BC	DE	IX	SP
DD-	09	19	29	39

Indicateurs :

S	Z		H		P/V	N	C
			●			○	●

H est positionné par le report du bit 11
C est positionné par le report du bit 15

Exemple : ADD IX, SP



Avant :

IX	0000
SP	3021

Après :

IX	3021
SP	3021

ADD IY, rr Additionner IY et le registre double rr.

Fonction : IY ← IY + rr

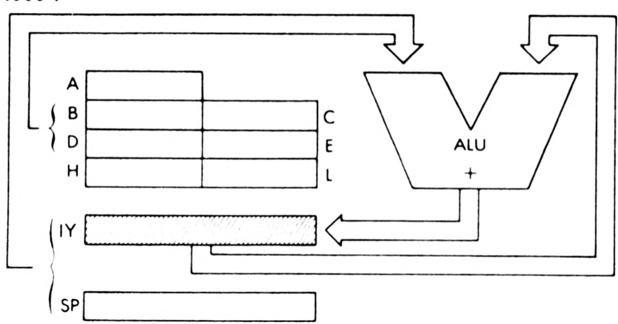
Format :

1	1	1	1	1	1	0	1	octet 1 : FD
0	0	r	r	1	0	0	1	octet 2

Description : Le contenu du registre IY est additionné au contenu du registre double spécifié et le résultat est rangé à nouveau dans IY. rr peut être n'importe lequel de :

BC – 00 IY – 10
DE – 01 SP – 11

Chemin des données :



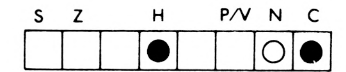
Durée : 4 cycles M ; 15 temps T : 7,5 usec @ 2 MHz

Mode d'adressage : Implicite

Codes :

rr:	BC	DE	IY	SP
FD-	09	19	29	39

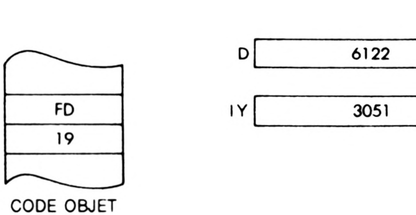
Indicateurs :

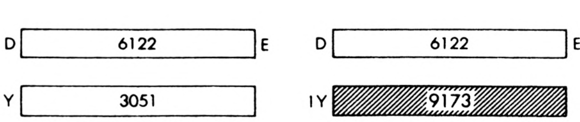


H est positionné par le report du bit 11
C est positionné par le report du bit 15

Exemple : ADD IY, DE

Avant : Après





AND s

Et logique entre l'accumulateur et l'opérande s.

Fonction : A ← A∧s

Format : s : peut être r, n, (HL), (IX + d), ou (IY + d)

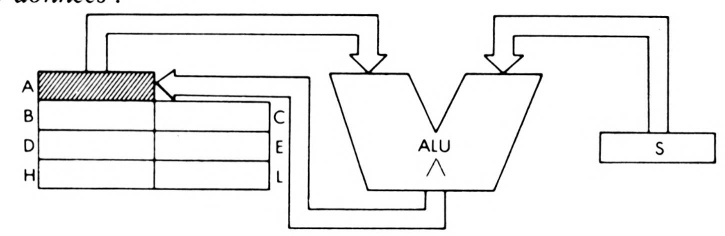
r	1	0	1	0	0	← r →			
n	1	1	1	0	0	1	1	0	octet 1 : E6
	← n →								octet 2 : valeur immédiate
(HL)	1	0	1	0	0	1	1	0	96
(IX + d)	1	1	0	1	1	1	0	1	octet 1 : DD
	1	0	1	0	0	1	1	0	octet 2 : 96
	← d →								octet 3 : déplacement
(IY + d)	1	1	1	1	1	1	0	1	octet 1 : FD
	1	0	1	0	0	1	1	0	octet 2 : 96
	← d →								octet 3 : déplacement

r peut être n'importe lequel de :

A – 111	E – 011
B – 000	H – 100
C – 011	L – 101
D – 010	

Description : Le ET logique de l'accumulateur et de l'opérande spécifié est effectué, et le résultat est rangé dans l'accumulateur. s est défini dans la description des instructions similaires ADD.

Chemin des données :



Durée :

s:	cycles M :	temps T :	usec @ 2 MHz:
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Mode d'adressage : r : implicite ; n : immédiat ; (HL) : indirect ; (IX + d), (IY + d) : indexé.

Codes : AND r

r	A	B	C	D	E	H	L
	A7	A0	A1	A2	A3	A4	A5

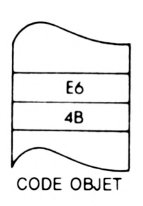
Indicateurs :

S	Z		H		P/V	N	C
●	●		1		●	○	○

Exemple : AND 4B

Avant : A 36

Après : A 02



BIT b, (HL) Test du bit b de l'emplacement mémoire d'adresse indirecte (HL).

Fonction : $Z \leftarrow \overline{(HL)_b}$

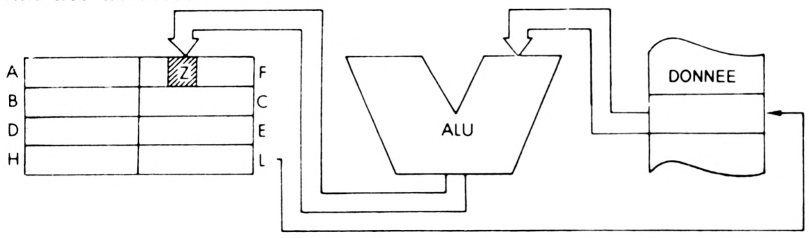
Format :

1	1	0	0	1	0	1	1	octet 1 : CB
0	1	←	b	→	1	1	0	octet 2

Description : Le bit spécifié de l'emplacement mémoire adressé par le contenu du registre double HL est testé et l'indicateur Z est positionné selon le résultat. b peut être n'importe lequel de :

0 – 000   4 – 100
1 – 001   5 – 101
2 – 010   6 – 110
3 – 011   7 – 111

Chemin des données :



Durée : 3 cycles M ; 12 temps T ; 6 usec @ 2 MHz

Mode d'adressage : Indirect.

Indicateurs :

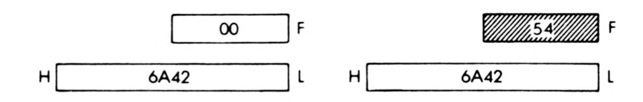
S	Z		H		P/V	N	C
?	●		1		?	0	

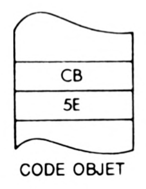
Codes :

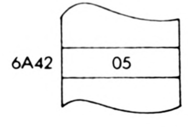
b:	0	1	2	3	4	5	6	7
CB-	46	4E	56	5E	66	6E	76	7E

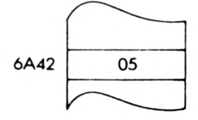
Exemple : BIT 3, (HL)

Avant : Après :





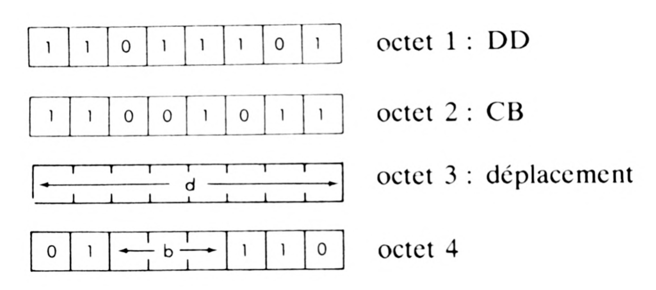




BIT b, (IX + d) Test du bit b de l'emplacement mémoire d'adresse indexée. (IX + d)

Fonction : $Z \leftarrow \overline{(IX + d)_b}$

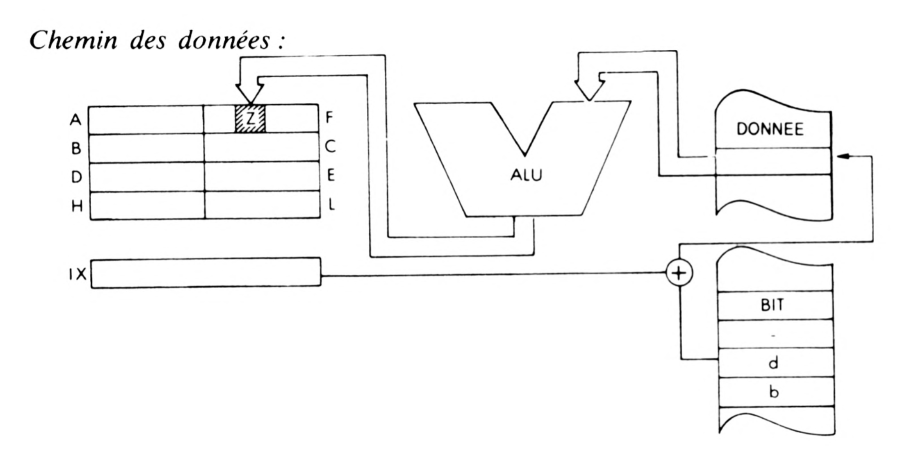
Format :



Description : Le bit spécifié de l'emplacement mémoire adressé par le contenu du registre IX plus le déplacement fourni est testé et l'indicateur Z est positionné selon le résultat. b peut être n'importe lequel de :

0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111

Chemin des données :



Durée : 5 cycles M ; 20 temps T : 10 usec @ 2 MHz

Mode d'adressage : Indexé.

Codes :

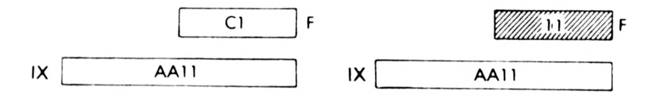
b:	0	1	2	3	4	5	6	7
DD-CB-d-	46	4E	56	5E	66	6E	76	7E

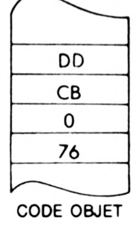
Indicateurs :

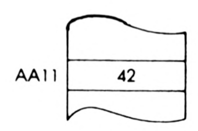
S	Z		H		P/V	N	C
?	●		1		?	0	

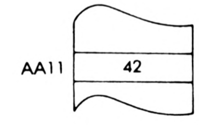
Exemple : BIT 6, (IX + 0)

Avant : Après :





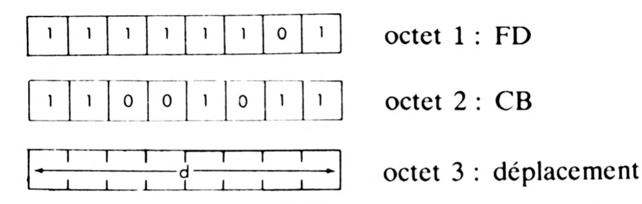




BIT b, (IY + d) Test du bit b de l'emplacement mémoire d'adresse indexée. (IY + d)

Fonction : $Z \leftarrow \overline{(IY + d)_b}$

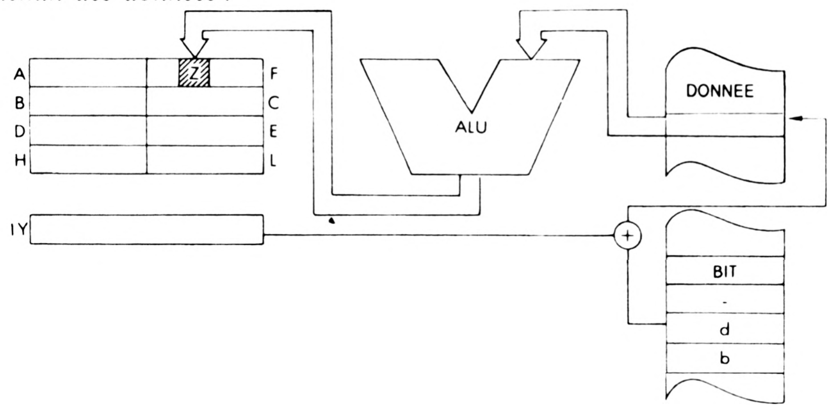
Format :



Description : Le bit spécifié de l'emplacement mémoire adressé par le contenu du registre IY plus le déplacement fourni est testé et l'indicateur Z est positionné selon le résultat. b peut être n'importe lequel de :

0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111


Chemin des données :



Durée : 5 cycles M ; 20 temps T ; 10 usec @ 2 MHz

Mode d'adressage : Indexé.

Codes : b :

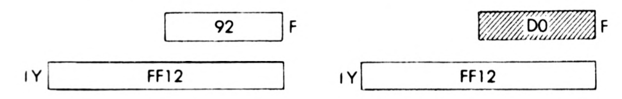
0	1	2	3	4	5	6	7
46	4E	56	5E	66	6E	76	7E

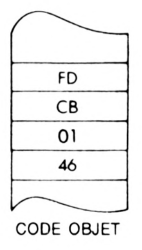
Indicateurs :

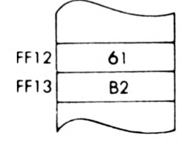
S	Z		H		P/V	N	C
?	●		1		?	0	

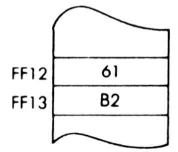
Exemple : **BIT 0, (IY + 1)**

Avant : Après :









BIT b, r

Test du bit b du registre r.

Fonction : $Z \leftarrow \overline{r_b}$

Format :

1	1	0	0	1	0	1	1	octet 1 : CB

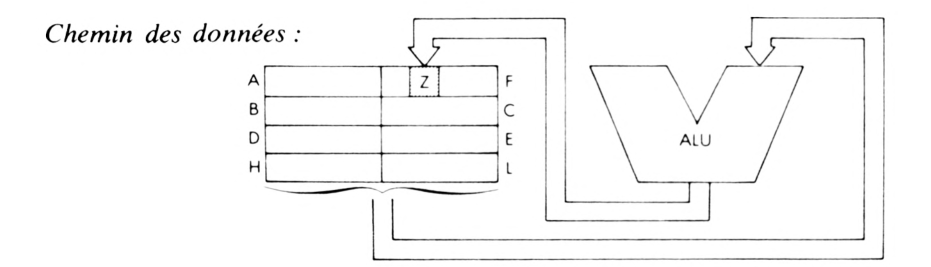
0	1	← b →			← r →			octet 2

Description :

Le bit spécifié du registre indiqué est testé et l'indicateur Z est positionné selon le résultat. b et r peuvent être n'importe lesquels de :

b :	0 – 000	4 – 100
	1 – 001	5 – 101
	2 – 010	6 – 110
	3 – 011	7 – 111
r:	A – 111	E – 011
	B – 000	H – 100
	C – 001	L – 101
	D – 010	

Chemin des données :



Durée : 2 cycles M ; 8 temps T ; 4 usec @ 2 MHz

Mode d'adressage : Implicite

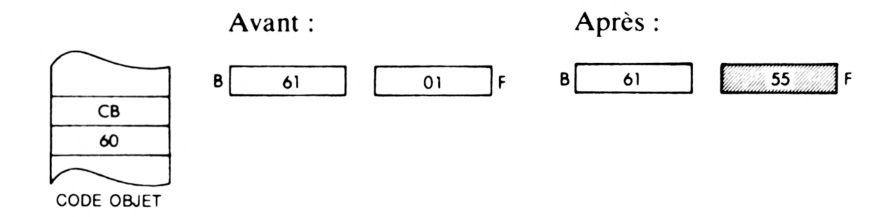
Codes :

b: \ r:	A	B	C	D	E	H	L
0	47	40	41	42	43	44	45
1	4F	48	49	4A	4B	4C	4D
2	57	50	51	52	53	54	55
3	5F	58	59	5A	5B	5C	5D
4	67	60	61	62	63	64	65
5	6F	68	69	6A	6B	6C	6D
6	77	70	71	72	73	74	75
7	7F	78	79	7A	7B	7C	7D

Indicateurs :

S	Z		H		P/V	N	C
?	●		1		?	0	

Exemple : **BIT 4, B**

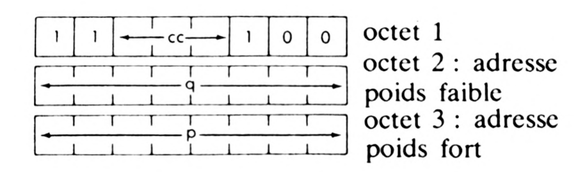


CALL cc, pq Appel conditionnel de sous-programme.

Fonction :

si cc vraie : (SP − 1) ← PC_{haut} ; (SP − 2) ← PC_{bas} ; SP ← SP − 2 ; PC ← pq
si cc fausse : PC ← PC + 3

Format :



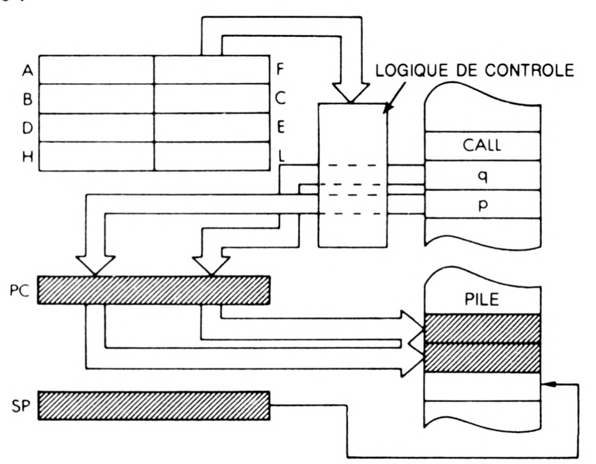
Description :

Si la condition est remplie, le contenu du compteur ordinal est empilé comme décrit pour les instructions PUSH. Ensuite, le contenu de l'emplacement mémoire suivant immédiatement le code opératoire est chargé dans le poids faible du PC et le contenu du second emplacement mémoire après le code opératoire est chargé dans la moitié de poids fort du PC. L'instruction suivante est cherchée à cette nouvelle adresse. Si la condition n'est pas remplie, l'adresse pq est ignorée et l'instruction suivante est exécutée. cc peut être n'importe laquelle de :

NZ − 000	PO − 100
Z − 001	PE − 101
NC − 010	P − 100
C − 011	M − 111

Une instruction RET peut être utilisée à la fin du sous-programme appelé pour restaurer PC.

Chemin des données :



Durée :

	cycles M :	*temps T :*	*usec @ 2 MHz*
condition vraie :	5	17	8.5
condition fausse :	3	10	5

Mode d'adressage : Immediat.

Codes :

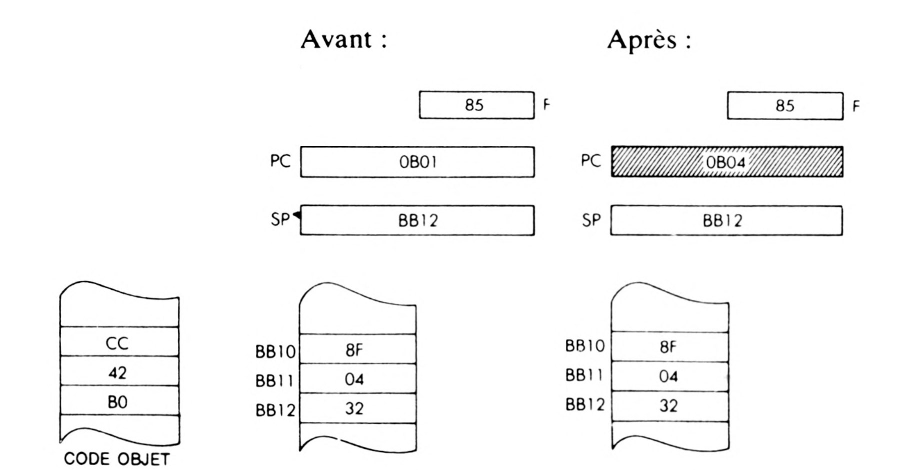
CC:	NZ	Z	NC	C	PO	PE	P	M	
	C4	CC	D4	DC	E4	EC	F4	FC	-q-p

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

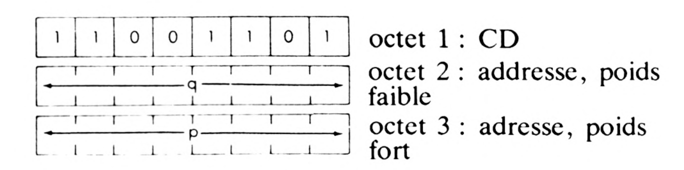
Exemple : CALL Z, B042



CALL pq Appel du sous-programme d'adresse pq.

Fonction : $(SP-1) \rightarrow PC_{haut}$; $(SP-2) \leftarrow PC_{bas}$; $SP \leftarrow SP - 2$; $PC \leftarrow pq$

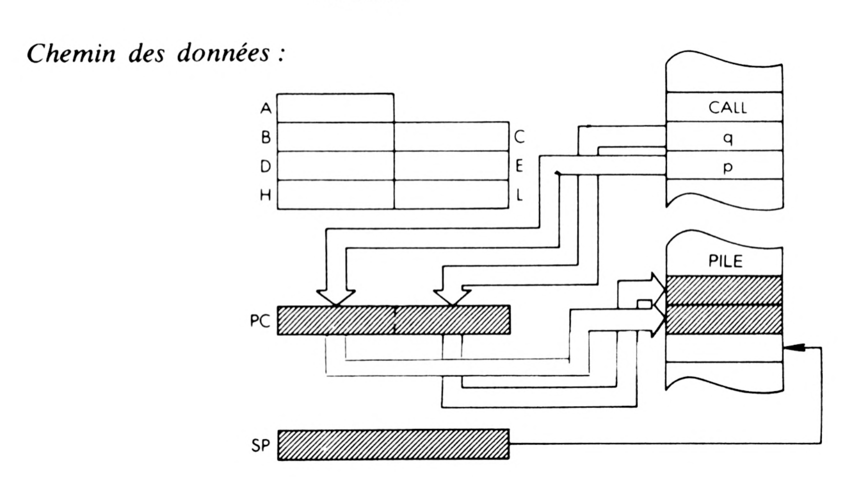
Format :



octet 1 : CD
octet 2 : addresse, poids faible
octet 3 : adresse, poids fort

Description : Le contenu du compteur ordinal est empilé comme décrit pour les instructions PUSH. Le contenu de l'emplacement mémoire suivant immédiatement le code opératoire est ensuite chargé dans la moitié de poids faible du PC et le contenu du second emplacement mémoire après le code opératoire est chargé dans la moitié de poids fort du PC. L'instruction suivante est cherchée à cette nouvelle adresse.

Chemin des données :

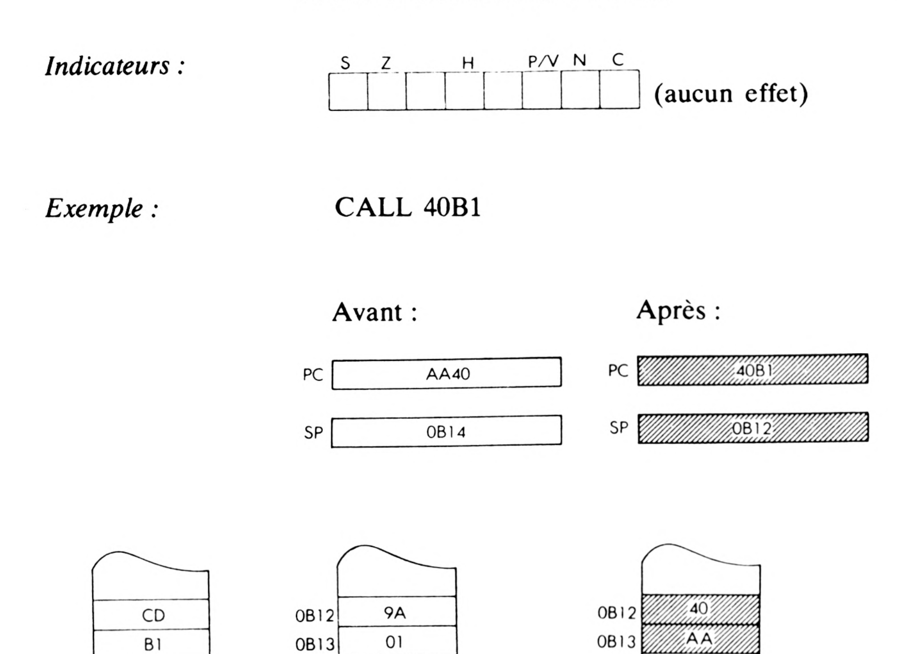


Durée : 5 cycles M ; 17 temps T : 8,5 usec @ 2 MHz

Mode d'adressage : Immediat.

Indicateurs :

Exemple : CALL 40B1



CCF Complémentation de l'indicateur de report.

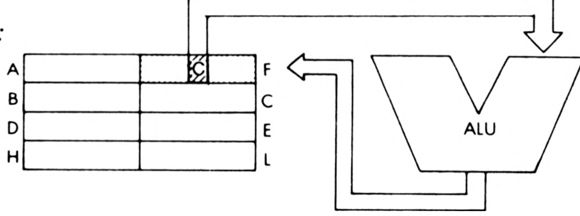
Fonction : $C \leftarrow \overline{C}$

Format :

0	0	1	1	1	1	1	1	3F

Description : L'indicateur de report est complimenté.

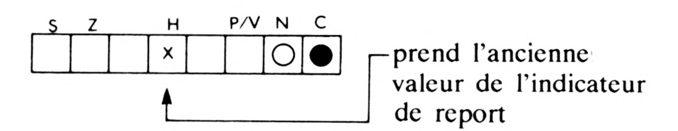

Chemin des données :



Durée : 1 cycle M ; 4 temps T : 2 usec @ 2 MHz

Mode d'adressage : Implicite

Indicateurs :



CP s

Comparaison de l'opérande s et de l'accumulateur.

Fonction : A − s

Format : s : peut être r, n, (HL), (IX + d), ou (IY + d).

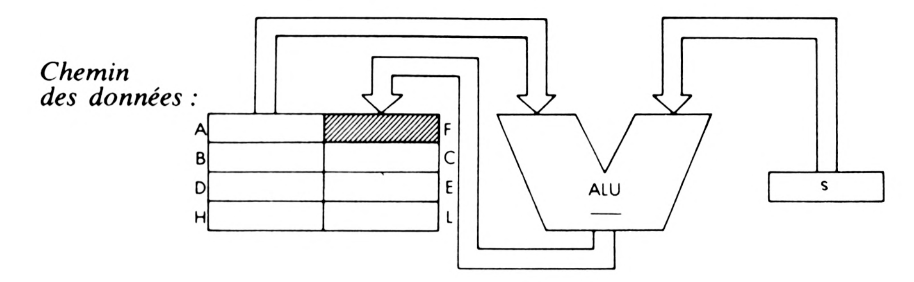
r	1 0 1 1 1 ←r→	
n	1 1 1 1 1 1 1 0	FE
	←n→	Octet 2 : valeur immédiate
(HL)	1 0 1 1 1 1 1 0	octet 1 : BE
(IX + d)	1 1 0 1 1 1 0 1	octet 1 : DD
	1 0 1 1 1 1 1 0	octet 2 : BE
	←d→	octet 3 : déplacement
(IY + d)	1 1 1 1 1 1 0 1	octet 1 : FD
	1 0 1 1 1 1 1 0	octet 2 : BE
	←d→	octet 3 : déplacement

r peut être n'importe lequel de :

A − 111	E − 011
B − 000	H − 100
C − 001	L − 101
D − 010	

Description : L'opérande spécifié est soustrait de l'accumulateur et le résultat n'est pas conservé. s est défini dans la description des instructions similaires ADD.

Chemin des données :

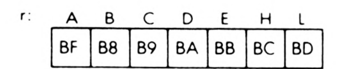


Durée :

s:	cycles M :	temps T :	usec @ 2 MHz:
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

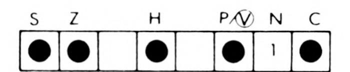
Mode d'adressage : r : implicite ; n : immediat ; (HL) : indirect ; (IX + d), (IY + d) : indexé

Codes : CP r :



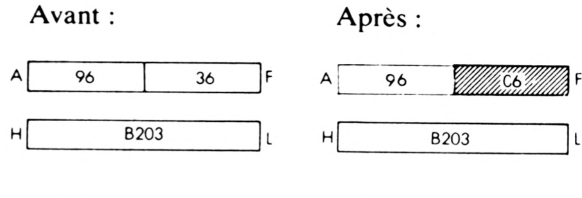
r:	A	B	C	D	E	H	L
	BF	B8	B9	BA	BB	BC	BD

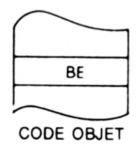
Indicateurs :

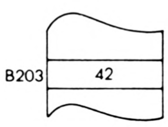


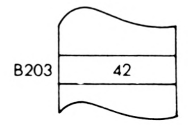
Exemple : CP (HL)

Avant : Après :









CPD

Comparaison avec décrémentation.

Fonction : A − (HL) ; HL ← HL − 1 ; BC ← BC − 1

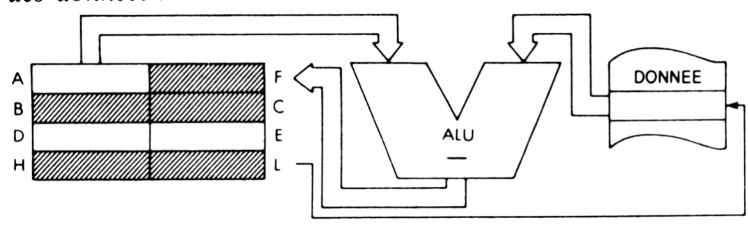
Format :

1	1	1	0	1	1	0	1	octet 1 : ED

1	0	1	0	1	0	0	1	octet 2 : A9

Description : Le contenu de l'emplacement mémoire adressé par le registre double HL est soustrait du contenu de l'accumulateur et le résultat n'est pas conservé. Ensuite chacun des deux registres doubles HL et BC est décrémenté.

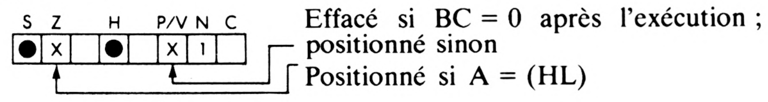
Chemin des données :



Durée : 4 cycles M ; 16 temps T ; 8 usec @ 2 MHz

Mode d'adressage : indirect.

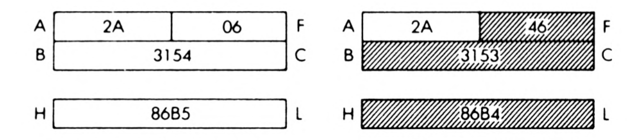
Indicateurs :

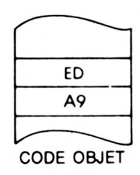


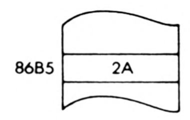
Exemple :

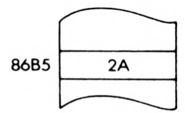
CPD

Avant : Après :









CPDR

Comparaison par bloc avec décrémentation.

Fonction :

A − (HL) ; HL ← HL − 1 ; BC ← BC − 1 ;
Répéter jusqu'à BC = 0 ou A = (HL)

Format :

1	1	1	0	1	1	0	1

octet 1 : ED

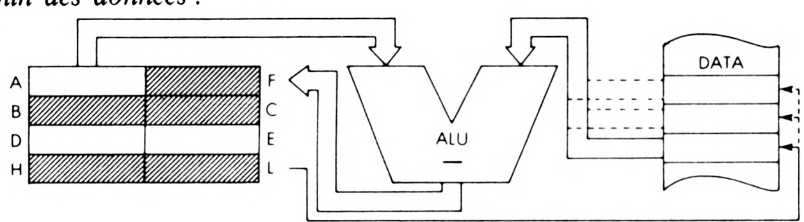
1	0	1	1	1	0	0	1

octet 2 : B9

Description :

Le contenu de l'emplacement mémoire adressé par le registre double HL est soustrait du contenu de l'accumulateur et le résultat n'est pas conservé. Ensuite chacun des deux registres doubles BC et HL est décrémenté. Si BC ≠ 0 et A ≠ (HL), le compteur ordinal est décrémenté de deux et l'instruction est réexécutée

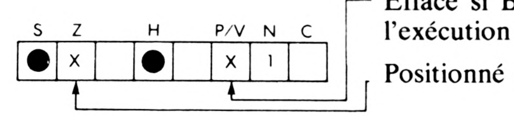
Chemin des données :



Durée :

BC = 0 ou A = (HL) : 4 cycles M ; 16 temps T : 8 usec @ 2 MHz
BC ≠ 0 et A ≠ [HL] : 5 cycles M ; 21 temps T ; 10.5 usec @ 2 MHz

Indicateurs :

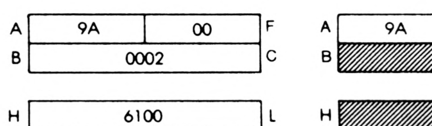


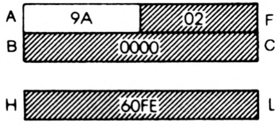
Effacé si BC = 0 après l'exécution ; positionné sinon

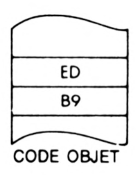
Positionné si A = (HL)

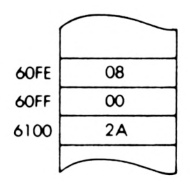
Exemple : CPDR

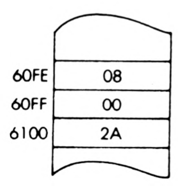
Avant : Après :











CPI

Comparaison avec incrémentation.

Fonction : A − (HL) ; HL ← HL + 1 ; BC ← BC − 1

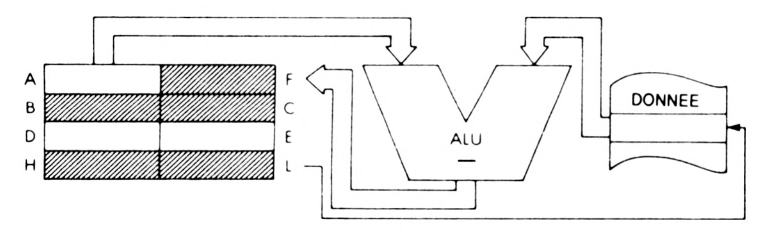
Format :

1	1	1	0	1	1	0	1	octet 1 : ED

1	0	1	0	0	0	0	1	octet 2 : A1

Description : Le contenu de l'emplacement mémoire adressé par le registre double HL est soustrait du contenu de l'accumulateur et le résultat n'est pas conservé. Le registre double HL est incrémenté et le registre double BC est décrémenté.

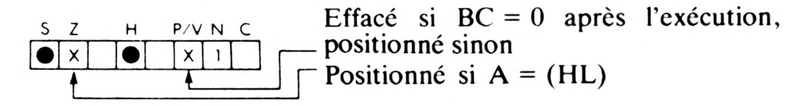
Chemin des données :



Durée : 4 cycles M ; 16 temps T : 8 usec @ 2 MHz

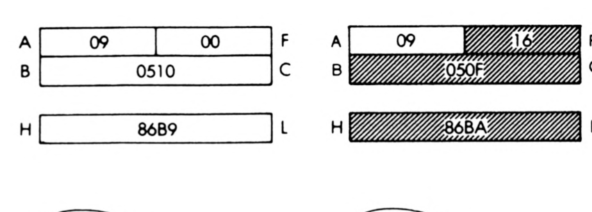
Mode d'adressage : indirect.

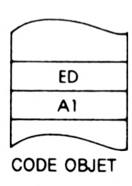
Indicateurs :

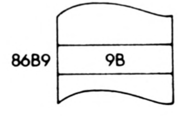


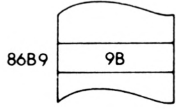
Exemple : CPI

Avant : Après :









CPIR

Comparaison par bloc avec incrémentation.

Fonction :

A − (HL) ; H ← HL + 1 ; BC ← BC − 1 ;
Répéter jusqu'à BC = 0 ou A = (HL)

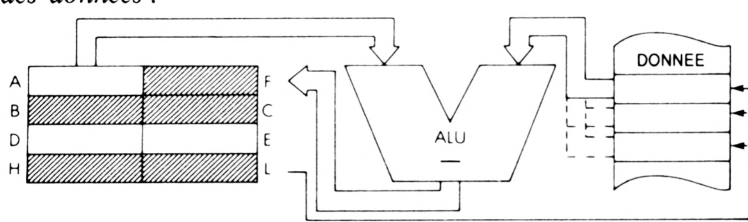
Format :

1	1	1	0	1	1	0	1	octet 1 : ED
1	0	1	1	0	0	0	1	octet 2 : B1

Description :

Le contenu de l'emplacement mémoire adressé par le registre double HL est soustrait du contenu de l'accumulateur et le résultat n'est pas conservé. Ensuite le registre double HL est incrémenté et le registre double BC est décrémenté. Si BC ≠ 0 et A ≠ (HL), alors le compteur ordinal est décrémenté de deux et l'instruction est réexécutée.

Chemin des données :

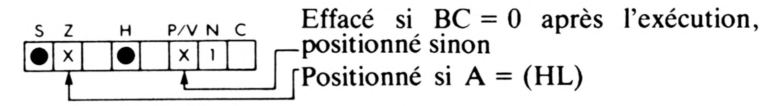


Durée :

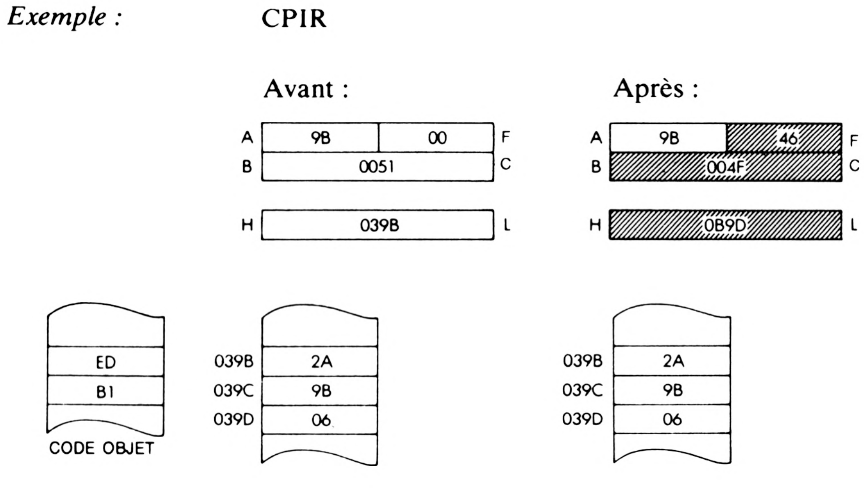
BC = 0 ou A = [HL] : 4 cycles M ; 16 temps T ; 8 usec @ 2 MHz
BC ≠ 0 et A ≠ [HL] : 5 cycles M ; 21 temps T ; 10,5 usec @ 2 MHz

Mode d'adressage : indirect.

Indicateurs :



Exemple : CPIR



CPL

Complémentation de l'accumulateur.

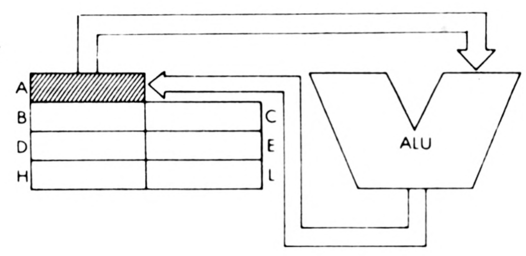
Fonction : $A \leftarrow \bar{A}$

Format :

0	0	1	0	1	1	1	1	2F

Description : Le contenu de l'accumulateur est complémenté, ou inversé, et le résultat est rangé à nouveau dans l'accumulateur (complément à un).

Chemin des données :



Durée : 1 cycle M ; 4 temps T ; 2 usec @ 2 MHz

Mode d'adressage : Implicite.

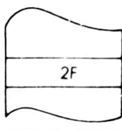
Indicateurs :

S	Z		H		P/V	N	C
			1			1	

Exemple : CPL

Avant : A 3D

Après : A C2



DAA Ajustement décimal de l'accumulateur.

Fonction : Voir ci-dessous

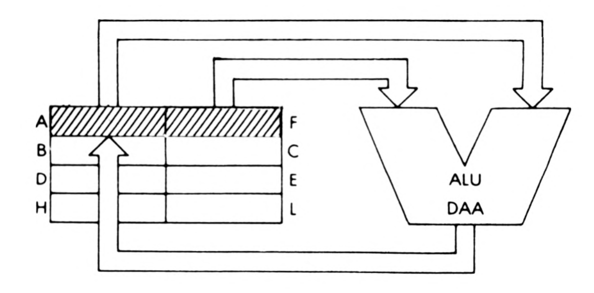
Format :

0	0	1	0	0	1	1	1	27

Description : Selon le registre d'indicateurs, cette instruction ajoute conditionnellement « 6 » au quartet de poids fort et/ou faible de l'accumulateur, pour la conversion DCB après les opérations arithmétiques.

N	C	valeur du quartet de poids fort	H	valeur du quartet de poids faible	# ajoutée à A	C après l'exécution
0	0	0-9	0	0-9	00	0
(ADD,	0	0-8	0	A-F	06	0
ADC,	0	0-9	1	0-3	06	0
INC)	0	A-F	0	0-9	60	1
	0	9-F	0	A-F	66	1
	0	A-F	1	0-3	66	1
	1	0-2	0	0-9	60	1
	1	0-2	0	A-F	66	1
	1	0-3	1	0-3	66	1
1	0	0-9	0	0-9	00	0
(SUB,	0	0-8	1	6-F	FA	0
SBC,	1	7-F	0	0-9	AO	1
DEC,	1	6-F	1	6-F	9A	1
NEG)						

Chemin des données :



Durée : 1 cycle M ; 4 temps T ; 2 usec @ 2 MHz

Mode d'adressage : Implicite

Indicateurs :

S	Z		H		(P)V	N	C
●	●		●		●		●

Exemple : DAA

Avant :

A		F
	B2	94

Après :

A		F
	18	05

CODE OBJET
27

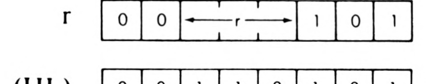
DEC m

Décrémentation de l'opérande m.

Fonction : m ← m − 1

Format : m : peut être r, (HL), (IX + d), (IY + d)

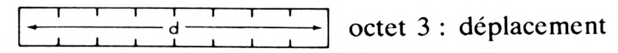
r



(HL) | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 35

(IX + d) | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | octet 1 : DD

| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | octet 2 : 35



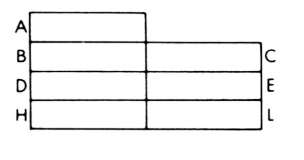
(IY + d) | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | octet 1 : FD

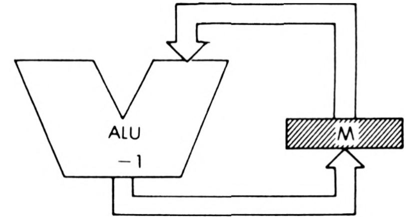
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | octet 2 : 35

| ← d → | octet 3 : déplacement

Description : Le contenu de l'emplacement adressé par l'opérande spécifié est décrémenté et rangé à nouveau dans cet emplacement ; m est défini dans la description des instructions similaires INC.

Chemin des données :



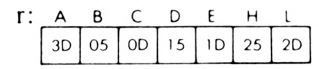


Durée :

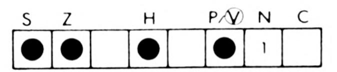
m:	cycles M :	temps T :	usec @ 2 MHz:
r	1	4	2
(HL)	3	11	5.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Mode d'adressage : r : implicite ; (HL) : indirect ; (IX + d), (IY + d) : indexé.

Codes : DEC r



Indicateurs :



Exemple : DEC C

Avant :

0F C

Après :



0D

CODE OBJET

DEC rr

Décrémentation du registre double rr.

Fonction : rr ← rr − 1

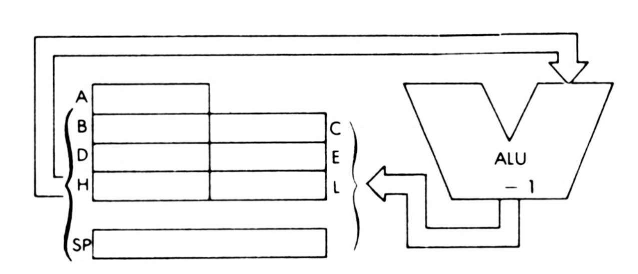
Format :

0	0	r	r	1	0	1	1

Description : Le contenu du registre double spécifié est décrémenté et le résultat est rangé à nouveau dans le registre double. rr peut être n'importe lequel de :

BC − 00 HL − 10
DE − 01 SP − 11

Chemin des données :



Durée : 1 cycle M ; 6 temps T ; 3 usec @ 2 MHz

Mode d'adressage : Implicite.

Codes : rr :

BC	DE	HL	SP
0B	1B	2B	3B

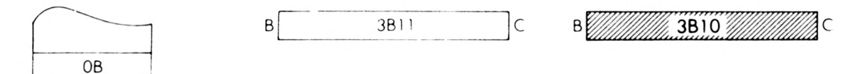
Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : DEC BC

Avant : Après :



CODE OBJET

DEC IX

Décrémentation de IX.

Fonction : IX ← IX − 1

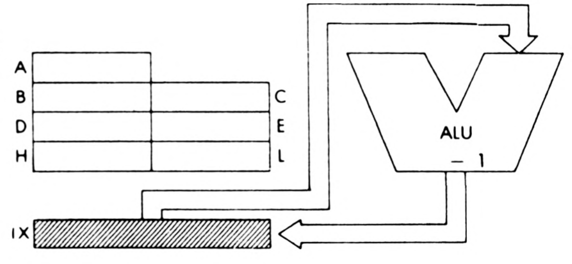
Format :

1	1	0	1	1	1	0	1	octet 1 : DD

0	0	1	0	1	0	1	1	octet 2 : 2B

Description : Le contenu du registre IX est décrémenté et le résultat est rangé à nouveau dans IX.

Chemin des données :



Durée : 2 cycles M ; 10 temps T ; 5 usec @ 2 MHz

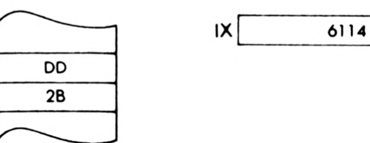
Mode d'adressage : Implicite

Indicateurs :

S	Z		H		P/V	N	C	
								(aucun effet)

Exemple : DEC IX

Avant : Après :



IX 6113

DEC IY

Décrémentation de IY.

Fonction : IY ← IY − 1

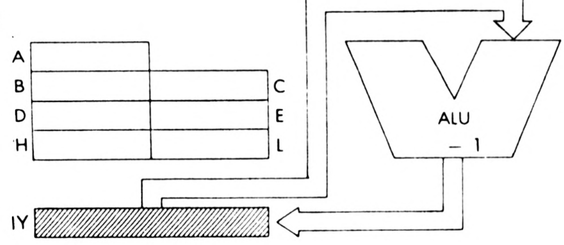
Format :

1	1	1	1	1	1	0	1	octet 1 : FD

0	0	1	0	1	0	1	1	octet 2 : 2B

Description : Le contenu du registre IY est décrémenté et le résultat est rangé à nouveau dans IY.

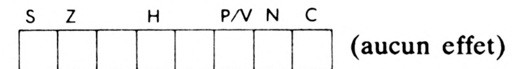
Chemin des données :



Durée : 2 cycles M ; 10 temps T ; 5 usec @ 2 MHz

Mode d'adressage : Implicite

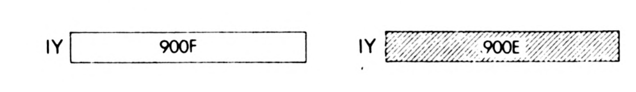
Indicateurs :



(aucun effet)

Exemple : DEC IY

Avant : Après :



FD
2B
CODE OBJET

DI Interdiction des interruptions.

Fonction : IFF ← 0

Format :

1	1	1	1	0	0	1	1	F3

Description : Les bascules d'interruption sont mises à zéro, interdisant ainsi toutes les interruptions masquables. Une interruption masquable peut être interdite durant son exécution par DI. Elle est réautorisée par l'instruction EI.

Durée : 1 cycle M ; 4 temps T ; 2 usec @ 2 MHz

Mode d'adressage : Implicite

Indicateurs :

S	Z		H		P/V	N	C	
								(aucun effet)

DJNZ e Décrémentation de B et saut relatif de e si non nul.

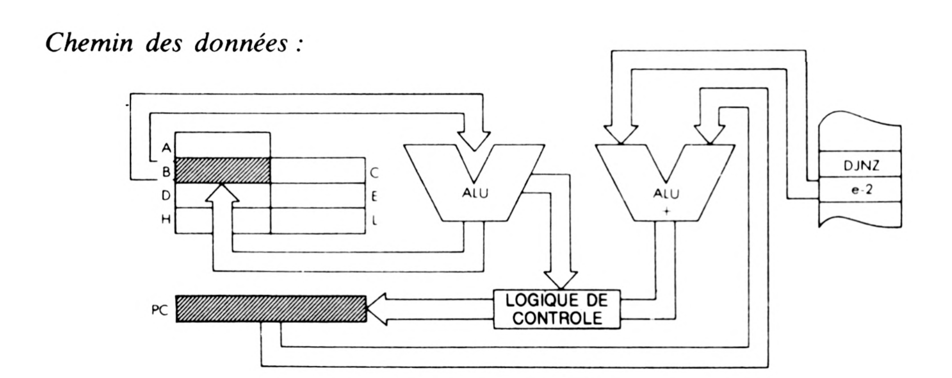
Fonction : B ← b − 1 ; si B ≠ 0 : PC ← PC + e

Format :

0	0	0	1	0	0	0	0	octet 1 : 10
← e-2 →								octet 2 : déplacement

Description : Le registre B est décrémenté. Si le résultat n'est pas zéro, le déplacement immédiat est ajouté au compteur ordinal, en utilisant l'arithmétique en complément à deux de façon à permettre à la fois des sauts en amont et en aval. Le déplacement est ajouté à la valeur de PC + 2 (après le saut). Ainsi le déplacement réel va de − 126 à + 129 octets. L'assembleur soustrait automatiquement 2 du déplacement source pour générer le code hexa.

Chemin des données :



Durée : B ≠ 0 : 3 cycles M ; 13 temps T ; 6,5 usec @ 2 MHz.

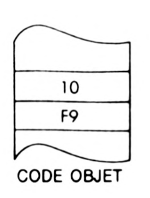
B = 0 : 2 cycles M ; 8 temps T ; 4 usec @ 2 MHz.

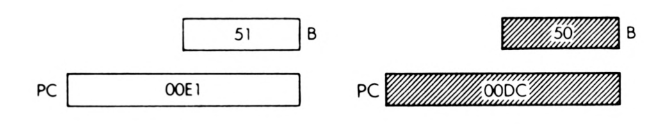
Mode d'adressage : Immediat.

Indicateurs : S Z H P/V N C (aucun effet)

Exemple : DJNZ $ − 5 ($ = PC courant)

Avant : Après :





EI

Autorisation des interruptions.

Fonction : IFF ← 1

Format :

1	1	1	1	1	0	1	1

FB

Description : Les bascules d'interruption sont mises à 1, autorisant ainsi les interruptions masquables après l'exécution de l'instruction suivant l'instruction EI. Dans l'intervalle les interruptions masquables sont interdites.

Durée : 1 cycle M ; 4 temps T ; 2 usec @ 2 MHz

Mode d'adressage : Implicite

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : Une séquence habituelle de fin d'une routine d'interruption est :

EI
RETI

L'interruption masquable est réautorisée après l'exécution de RETI

EX AF, AF'

Echange de l'accumulateur et des indicateurs avec les registres auxiliaires.

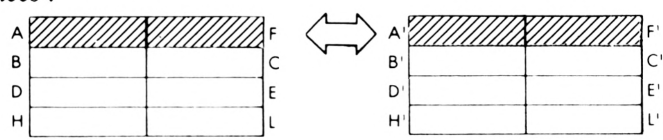
Fonction : AF ↔ AF'

Format :

0	0	0	0	1	0	0	0	08

Description : Les contenus de l'accumulateur et du registre d'état sont échangés avec les contenus des accumulateurs et registre d'état auxiliaires.

Chemin des données :



Durée : 1 cycle M ; 4 temps T ; 2 usec @ 2 MHz

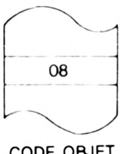
Mode d'adressage : Implicite

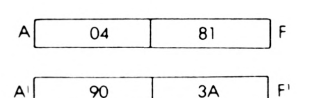
Indicateurs :

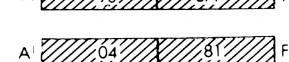
S	Z		H		P/V	N	C	
								(aucun effet)

Exemple : EX AF, AF'

Avant : Après :







EX DE, HL Echange des registres HL et DE.

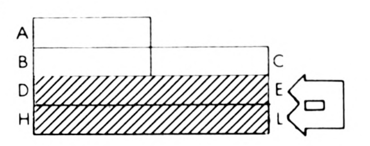
Fonction : DE ↔ HL

Format :

1	1	1	0	1	0	1	1	EB

Description : Les contenus des registres doubles DE et HL sont échangés.

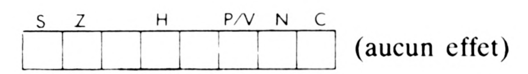
Chemin des données :



Durée : 1 cycle M ; 4 temps T ; 2 usec @ 2 MHz

Mode d'adressage : Implicite

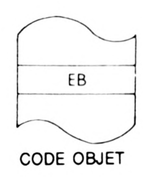
Indicateurs :



(aucun effet)

Exemple : EX DE, HL

Avant : Après :



D A4E6 E
H 9604 L



EX (SP), HL Echange de HL avec le sommet de la pile.

Fonction : (SP) ↔ L ; (SP + 1) ↔ H

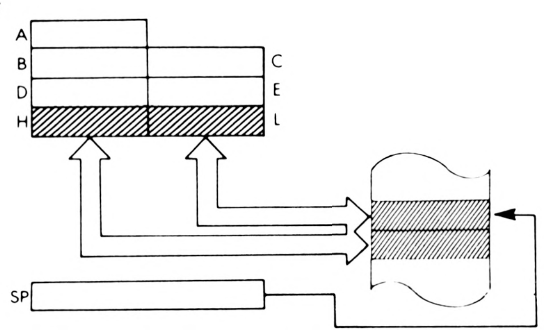
Format :

1	1	1	0	0	0	1	1

E3

Description : Le contenu du registre L est échangé avec le contenu de l'emplacement mémoire adressé par le pointeur de pile. Le contenu du registre H est échangé avec le contenu de l'emplacement mémoire suivant immédiatement celui adressé par le pointeur de pile.

Chemin des données :



Durée : 5 cycles M ; 19 temps T ; 9,5 usec @ 2 MHz

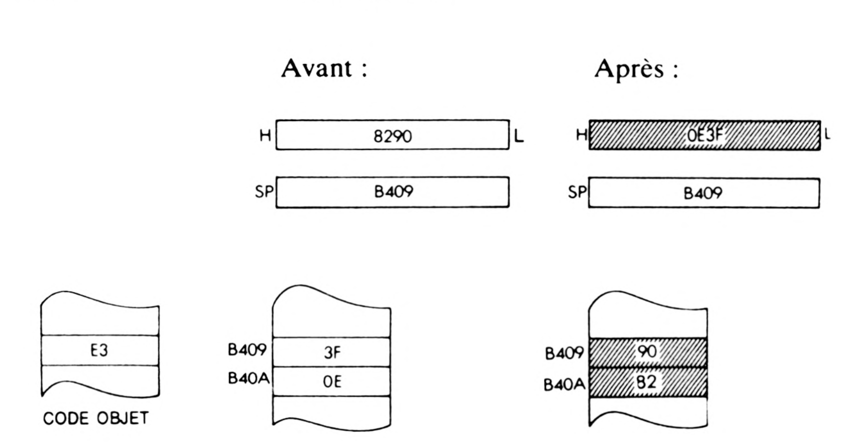
Mode d'adressage : Indirect.

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : EX (SP), HL



EX (SP), IX

Echange de IX avec le sommet de la pile.

Fonction :

$(SP) \leftrightarrow IX_{bas}$; $(SP + 1) \leftrightarrow IX_{haut}$

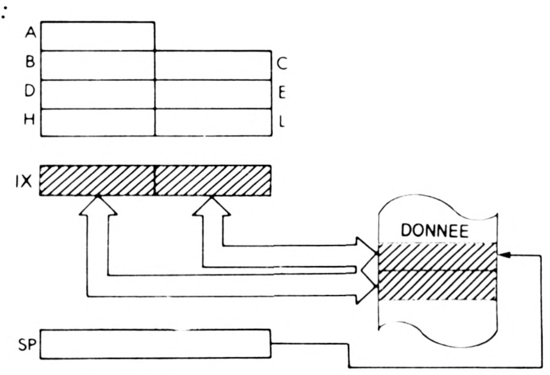
Format :

1	1	0	1	1	1	0	1	octet 1 : DD
1	1	1	0	0	0	1	1	octet 2 : E3

Description :

Le contenu du poids faible du registre IX est échangé avec le contenu de l'emplacement mémoire adressé par le pointeur de pile. Le contenu du poids fort du registre IX est échangé avec le contenu de l'emplacement mémoire suivant immédiatement celui adressé par le pointeur de pile.

Chemin des données :



Durée :

6 cycles M ; 23 temps T ; 11,5 usec @ 2 MHz

Mode d'adressage :

Indirect.

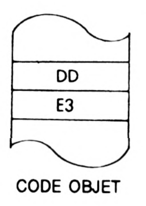
Indicateurs :

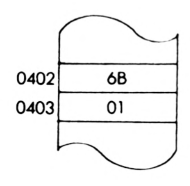
S	Z		H		P/V	N	C

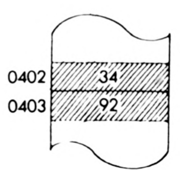
(aucun effet)

Exemple : EX (SP), IX

	Avant :	Après :
IX	9234	016B
SP	0402	0402







EX (SP), IY Echange de IY avec le sommet de la pile.

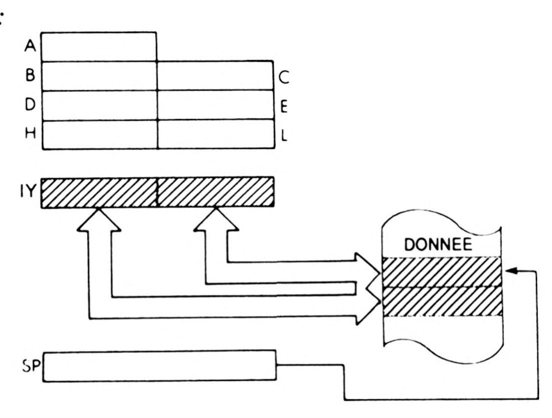
Fonction : (SP) ↔ IY_{bas} ; (SP + 1) ↔ IY_{haut}

Format :

1	1	1	1	1	1	0	1	octet 1 : FD
1	1	1	0	0	0	1	1	octet 2 : E3

Description : Le contenu du poids faible du registre IY est échangé avec le contenu de l'emplacement mémoire adressé par le pointeur de pile. Le contenu du poids fort du registre IY est échangé avec le contenu de l'emplacement mémoire suivant immédiatement celui adressé par le pointeur de pile.

Chemin des données :



Durée : 6 cycles M ; 23 temps T ; 11,5 usec @ 2 MHz

Mode d'adressage : Indirect.

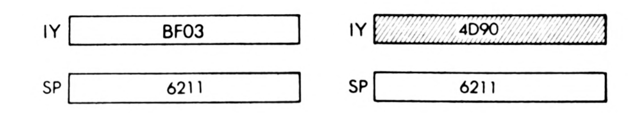
Indicateurs :

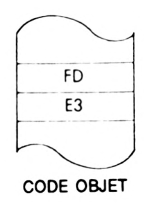
S	Z		H		P/V	N	C

(aucun effet)

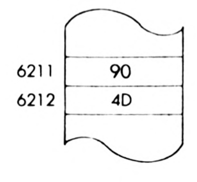
Exemple : EX (SP), IY

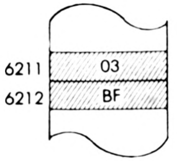
Avant : Après :





CODE OBJET





EXX Echange avec les registres auxiliaires.

Fonction : BC ↔ BC′ ; DE ↔ DE′ ; HL ↔ HL′

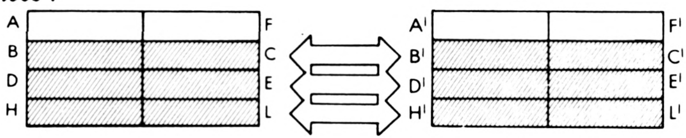
Format :

1	1	0	1	1	0	0	1

D9

Description : Les contenus des registres d'usage général sont échangés avec les contenus des registres auxiliaires correspondants.

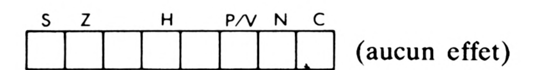
Chemin des données :



Durée : 1 cycle M ; 4 temps T ; 2 usec @ 2 MHz

Mode d'adressage : Implicite

Indicateurs :

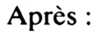


(aucun effet)

Exemple : EXX

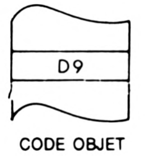
Avant :

Après :

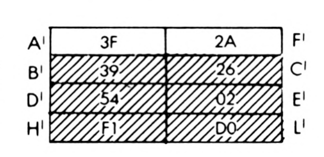


A	04	2B	F
B	39	26	C
D	54	02	E
H	F1	D0	L

A	04	2B	F
B	8C	00	C
D	93	D0	E
H	4F	E3	L



A'	3F	2A	F'
B'	8C	00	C'
D'	93	D0	E'
H'	4F	E3	L'



HALT Halte de l'unité centrale.

Fonction : Suspension du fonctionnement de l'unité centrale

Format :

0	1	1	1	0	1	1	0	76

Description : La CPU suspend son fonctionnement et exécute des NOP pour continuer les cycles de rafraîchissement de la mémoire, jusqu'à recevoir une interruption ou une réinitialisation.

Durée : 1 cycle M ; 4 temps T ; 2 usec @ 2 MHz + un nombre indéfini de NOP

Mode d'adressage : Implicite

Indicateurs :

S	Z		H		P/V	N	C	
								(aucun effet)

IM 0

Sélection du mode d'interruption 0.

Fonction : Contrôle interne des interruptions

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
0	1	0	0	0	1	1	0	octet 2 : 46

Description : Sélection du mode d'interruption 0. Dans ce mode, l'élément qui interrompt doit placer une instruction à exécuter sur le bus de données ; le premier octet de l'instruction doit arriver pendant le cycle d'acquittement de l'interruption.

Durée : 2 cycles M ; 8 temps T ; 4 usec @ 2 MHz

Mode d'adressage : Implicite

Indicateurs :

S	Z		H		P/V	N	C	
								(aucun effet)

IM 1

Sélection du mode d'interruption 1.

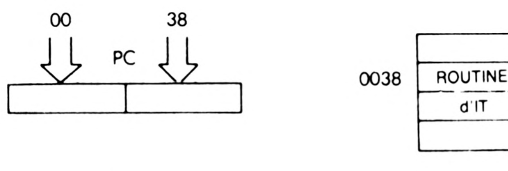
Fonction : Contrôle interne des interruptions

Format :

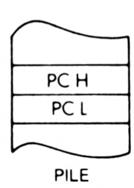
1	1	1	0	1	1	0	1	octet 1 : ED
0	1	0	1	0	1	1	0	octet 2 : 56

Description : Sélection du mode d'interruption 1. Une instruction RST 0038H sera exécutée lorsqu'une interruption surviendra.

Chemin des données :



lors de l'interruption



Durée : 2 cycles M ; 8 temps T ; 4 usec @ 2 MHz

Mode d'adressage : Implicite

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

IM 2

Sélection du mode d'interruption 2.

Fonction : Contrôle interne des interruptions

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
0	1	0	1	1	1	1	0	octet 2 : 5E

Description : Sélection du mode d'interruption 2. Lorsqu'une interruption survient, le périphérique doit fournir un octet de donnée qui est utilisé comme partie basse d'une adresse. La partie haute de cette adresse est donnée par le contenu du registre I. Cette adresse pointe sur une seconde adresse rangée en mémoire, qui est chargée dans le compteur ordinal et l'exécution commence.

Durée : 2 cycles M ; 8 temps T ; 4 usec @ 2 MHz

Mode d'adressage : Implicite

Indicateurs :

S	Z		H		P/V	N	C	
								(aucun effet)

IN r, (C)

Chargement du registre r à partir du port (C).

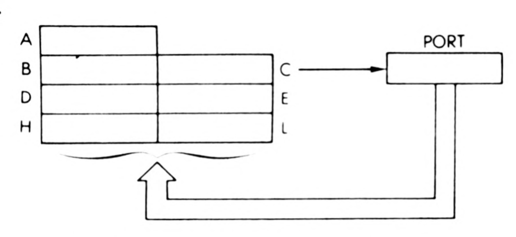
Fonction : r ← (C)

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
0	1	←	r	→	0	0	0	octet 2

Description : L'organe périphérique adressé par le contenu du registre C est lu et le résultat est chargé dans le registre spécifié.
C fournit les bits A0 à A7 du bus adresse.
B fournit les bits A8 à A15.

Chemin des données :



r peut être n'importe lequel de :

A – 111
B – 000
C – 001
D – 010
E – 011
H – 100
L – 101

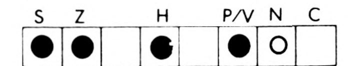
Durée : 3 cycles M ; 12 temps T ; 6 usec @ 2 MHz

Mode d'adressage : Externe.

Codes :

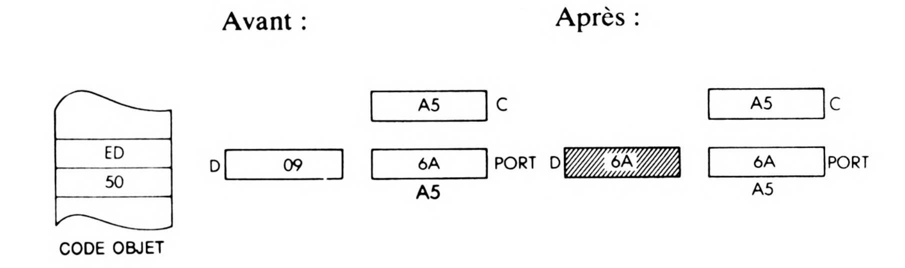
r:	A	B	C	D	E	H	L
ED	78	40	48	50	58	60	68

Indicateurs :



Il est important de noter que IN A, (N) n'a aucun effet sur les indicateurs, alors que IN r, (C) en a.

Exemple : IN D, (C)



IN A, (N) Chargement de l'accumulateur à partir du port N.

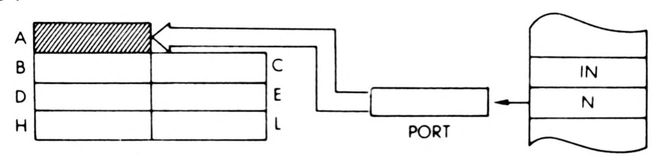
Fonction : A ← (N)

Format :

1	1	0	1	1	0	1	1	octet 1 : DB
←			N				→	octet 2 : adresse du port

Description : L'organe périphérique N est lu et le résultat est chargé dans l'accumulateur.
Le littéral N est placé sur les lignes A0 à A7 du bus adresse, A fournit les bits A8 à A15.

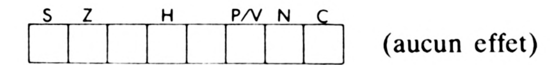
Chemin des données :



Durée : 3 cycles M ; 11 temps T ; 5,5 usec @ 2 MHz

Mode d'adressage : Externe.

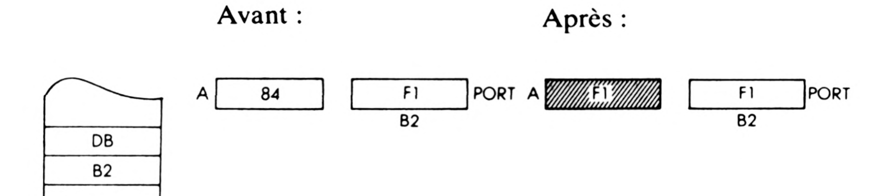
Indicateurs :



(aucun effet)

Exemple : IN A, (B2)

Avant : Après :



INC r

Incrémentation du registre r.

Fonction : r ← r + 1

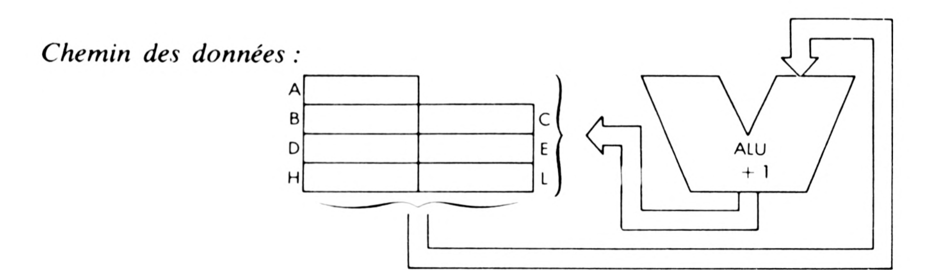
Format :

0	0	← r →	1	0	0

Description : Le contenu du registre spécifié est incrémenté. r peut être n'importe lequel de :

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Chemin des données :



Durée : 1 cycle M ; 4 temps T ; 2 usec @ 2 MHz

Mode d'adressage : Implicite

Codes :

r:	A	B	C	D	E	H	L
	3C	04	0C	14	1C	24	2C

Indicateurs :

S	Z		H		P/V	N	C
●	●		●		●	○	

Exemple : INC D

14

CODE OBJET



INC rr

Incrémentation du registre double rr.

Fonction : rr ← rr + 1

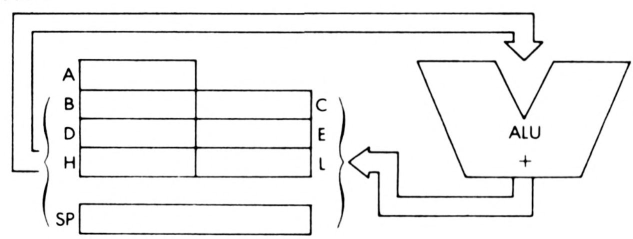
Format :

0	0	r	r	0	0	1	1

Description : Le contenu du registre double spécifié est incrémenté et le résultat est rangé à nouveau dans le registre double. rr peut être n'importe lequel de :

BC – 00 HL – 10
DE – 01 SP – 11

Chemin des données :



Durée : 1 cycle M ; 6 temps T ; 3 usec @ 2 MHz

Mode d'adressage : Implicite

Codes :

rr:

BC	DE	HL	SP
03	13	23	33

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

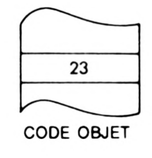
Exemple : INC HL

Avant :

H | 0B14 | L

Après :

H | 0B15 | L



INC (HL)

Incrémentation de l'emplacement mémoire d'adresse indirecte (HL).

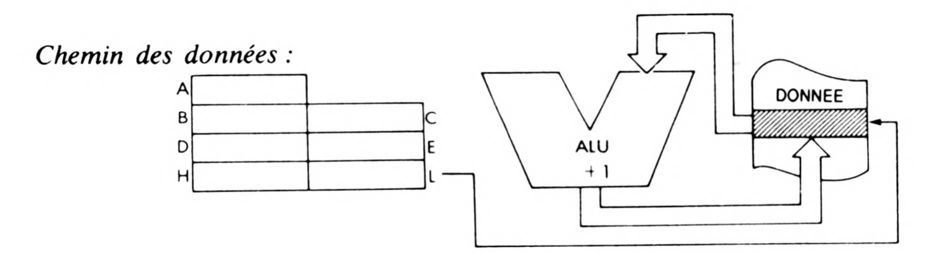
Fonction : (HL) ← (HL) + 1

Format :

0	0	1	1	0	1	0	0	34

Description : Le contenu de l'emplacement mémoire adressé par le registre double HL est incrémenté et rangé à nouveau à cet emplacement.

Chemin des données :



Durée : 3 cycles M ; 11 temps T ; 5,5 usec @ 2 MHz

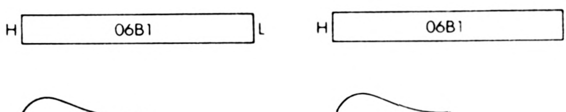
Mode d'adressage : Indirect.

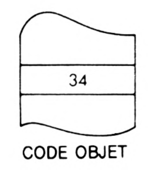
Indicateurs :

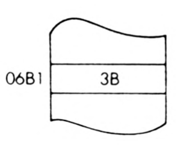
S	Z		H		P/V	N	C
●	●		●		●	○	

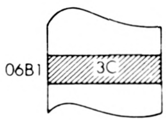
Exemple : INC (HL)

Avant : Après :







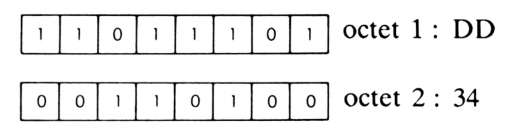


INC (IX + d)

Incrémentation de l'emplacement mémoire d'adresse indexée (IX + d).

Fonction : (IX + d) ← (IX + d) + 1

Format :



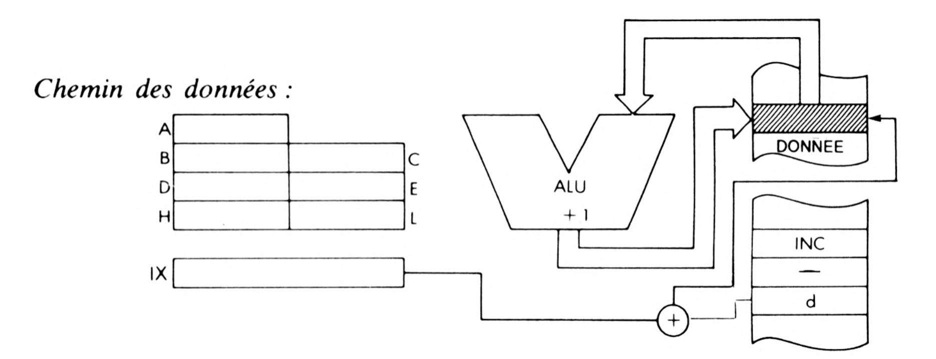
octet 1 : DD

octet 2 : 34

octet 3 : déplacement

Description : Le contenu de l'emplacement mémoire adressé par le contenu du registre IX plus le déplacement fourni est incrémenté et rangé à nouveau à cet emplacement.

Chemin des données :



Durée : 6 cycles M ; 23 temps T ; 11,5 usec @ 2 MHz

Mode d'adressage : Indexé.

Indicateurs :

S	Z		H		P/V	N	C
●	●		●		●	○	

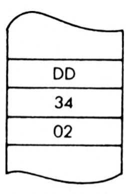
Exemple : INC (IX + 2)

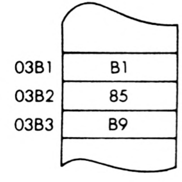
Avant :

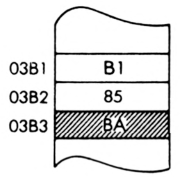
IX 03B1

Après :

IX 03B1







INC (IY + d) Incrémentation de l'emplacement mémoire d'adresse indexée (IY + d).

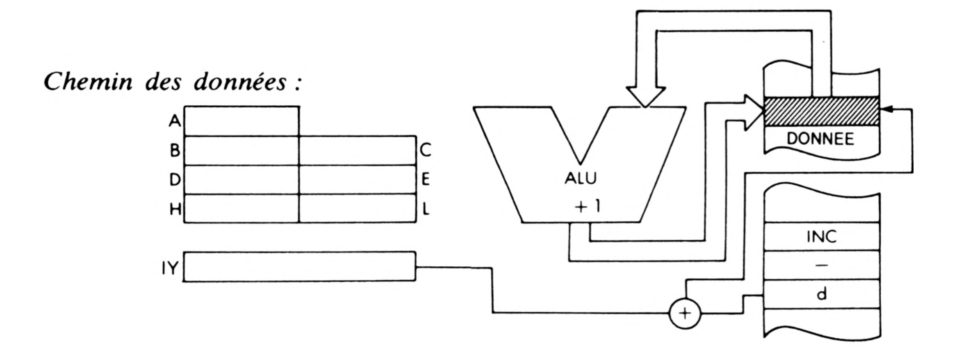
Fonction : (IY + d) ← (IY + d) + 1

Format :

1	1	1	1	1	1	0	1	octet 1 : FD
0	0	1	1	0	1	0	0	octet 2 : 34
←			d				→	octet 3 : déplacement

Description : Le contenu de l'emplacement mémoire adressé par le contenu du registre IY plus le déplacement fourni est incrémenté puis rangé à nouveau dans cet emplacement.

Chemin des données :



Durée : 6 cycles M ; 23 temps T ; 11,5 usec @ 2 MHz

Mode d'adressage : Indexé.

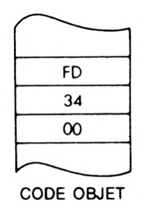
Indicateurs :

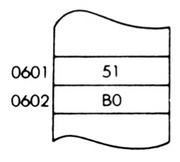
S	Z		H		P/V	N	C
●	●		●		●	○	

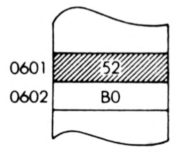
Exemple : INC (IY + 0)

Avant : Après :

IY 0601 IY 0601







INC IX — Incrémentation de IX.

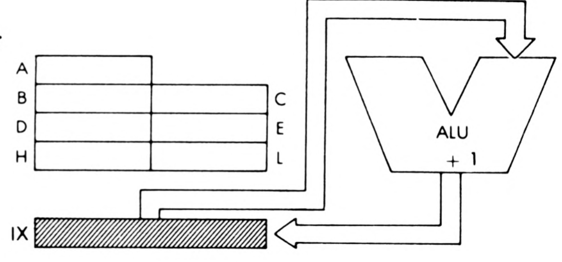
Fonction : IX ← IX + 1

Format :

1	1	0	1	1	1	0	1	octet 1 : DD
0	0	1	0	0	0	1	1	octet 2 : 23

Description : Le contenu du registre IX est incrémenté et le résultat est rangé à nouveau dans IX.

Chemin des données :



Durée : 2 cycles M ; 10 temps T ; 5 usec @ 2 MHz

Mode d'adressage : Implicite

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : INC IX

Avant : IX B1B0

Après : IX B1B1

DD
23

CODE OBJET



INC IY

Incrémentation de IY.

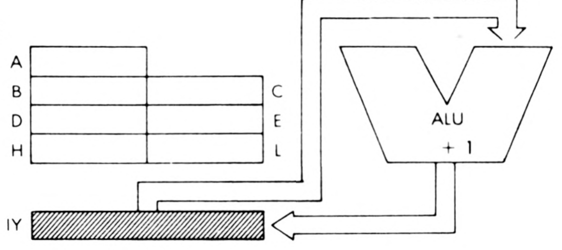
Fonction : IY ← IY + 1

Format :

1	1	1	1	1	1	0	1	octet 1 : FD
0	0	1	0	0	0	1	1	octet 2 : 23

Description : Le contenu du registre IY est incrémenté et le résultat est rangé à nouveau dans IY.

Chemin des données :



Durée : 2 cycles M ; 10 temps T ; 5 usec @ 2 MHz

Mode d'adressage : Implicite.

Indicateurs :

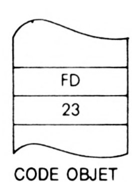
S	Z		H		P/V	N	C

(aucun effet)

Exemple : INC IY

Avant : Après :





CODE OBJET

IND Entrée avec décrémentation.

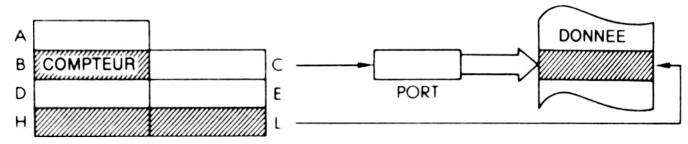
Fonction : (HL) ← (C) ; B ← B − 1 ; HL ← HL − 1

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
1	0	1	0	1	0	1	0	octet 2 : AA

Description : L'organe périphérique adressé par le registre C est lu et le résultat est chargé dans l'emplacement mémoire adressé par le registre double HL. Le registre B et le registre double HL sont ensuite décrémentés.

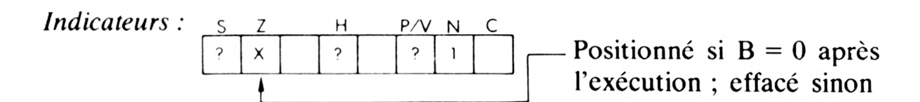
Chemin des données :



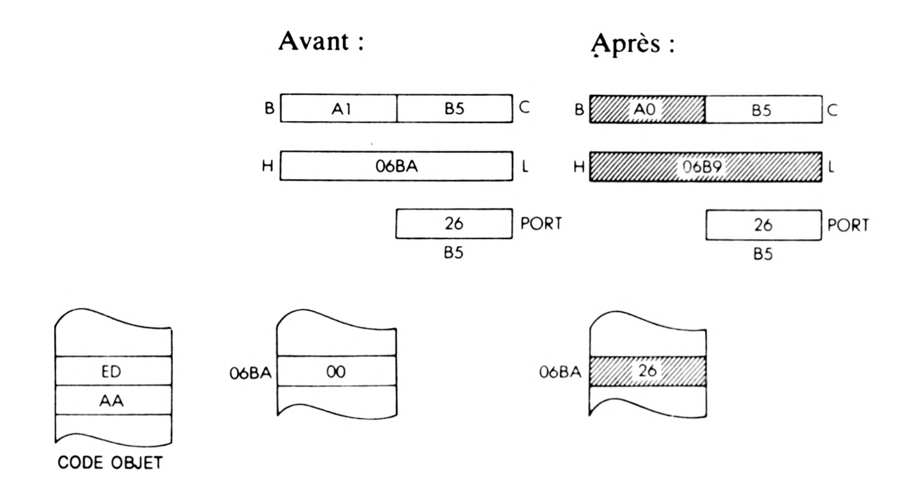
Durée : 4 cycles M ; 16 temps T ; 8 usec @ 2 MHz

Mode d'adressage : Externe

Indicateurs :



Exemple : IND


Avant :
Après :
B
A1
B5
C
H
06BA
L
26
PORT
B5
A0
06B9
ED
AA
CODE OBJET
06BA
00
26

INDR

Entrée par bloc avec décrémentation.

Fonction :

(HL) ← (C) ; B ← B − 1 ; HL ← HL − 1
Répéter jusqu'à B = 0

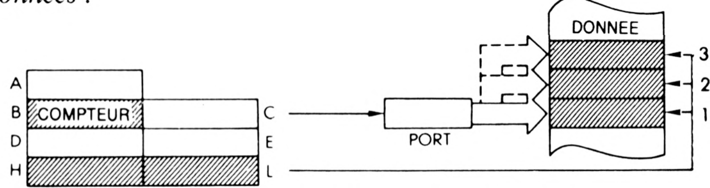
Format :

1	1	1	0	1	1	0	1	octet 1 : ED
1	0	1	1	1	0	1	0	octet 2 : BA

Description :

L'organe périphérique adressé par le registre C est lu et le résultat est chargé dans l'emplacement mémoire adressé par le registre double HL. Le registre B et le registre double HL sont ensuite décrémentés. Si le registre B est non nul, le compteur ordinal est décrémenté de 2 et l'instruction est réexécutée.

Chemin des données :



Durée :

B = 0 : 4 cycles M ; 16 temps T ; 8 usec @ 2 MHz.

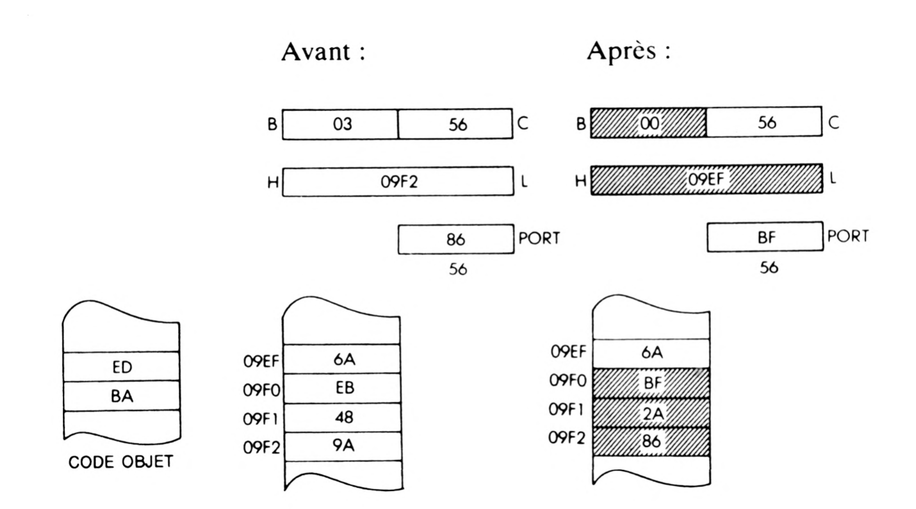
B ≠ 0 : 5 cycles M ; 21 temps T ; 10,5 usec @ 2 MHz.

Mode d'adressage : Externe

Indicateurs :

S	Z		H		P/V	N	C
?	1		?		?	1	

Exemple : INDR



INI Entrée avec incrémentation.

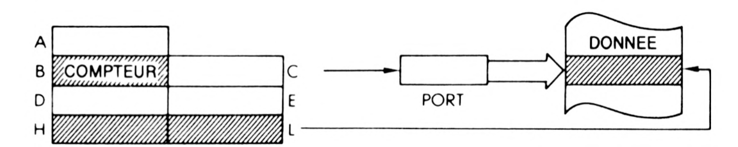
Fonction : (HL) ← (C) ; B ← B − 1 ; HL ← HL + 1

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
1	0	1	0	0	0	1	0	octet 2 : A2

Description : L'organe périphérique adressé par le registre C est lu et le résultat est chargé dans l'emplacement mémoire adressé par le registre double HL. Le registre B est décrémenté et le registre double HL est incrémenté. Le contenu de C est placé sur la moitié basse du bus adresse. Le contenu de B est placé sur la moitié haute. La sélection des ports d'entrée-sortie est généralement faite par C, c'est-à-dire A_0 à A_7. B sert de compte d'octets.

Chemin des données :



Durée : 4 cycles M ; 16 temps T ; 8 usec @ 2 MHz

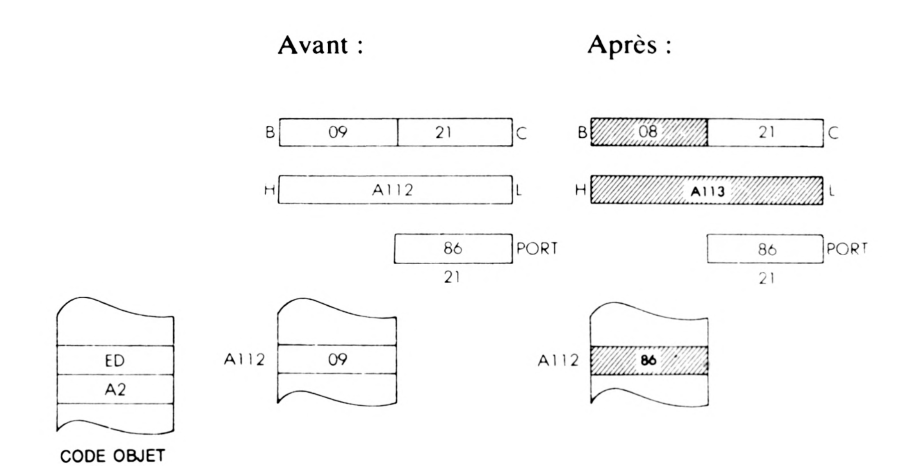
Mode d'adressage : Externe

Indicateurs :

S	Z		H		P/V	N	C
?	X		?		?	1	

Z est positionné si B = 0 après l'exécution, effacé sinon.

Exemple : **INI**


Avant :
Après :
B
09
21
C
H
A112
L
86
PORT
21
B
08
21
C
H
A113
L
86
PORT
21
ED
A2
CODE OBJET
A112
09
A112
86

INIR Entrée par bloc avec incrémentation.

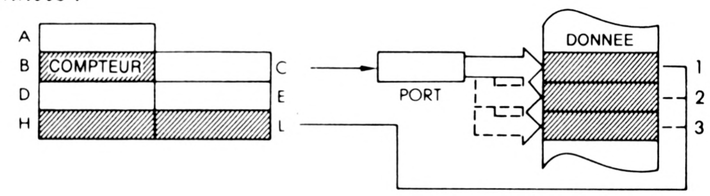
Fonction : (HL) ← (C) ; B ← B − 1 ; HL ← HL + 1 ; Répéter jusqu'à B = 0

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
1	0	1	1	0	0	1	0	octet 2 : B2

Description : L'organe périphérique adressé par le registre C est lu et le résultat est chargé dans l'emplacement mémoire adressé par le registre double HL. Le registre B est décrémenté et le registre double HL est incrémenté. Si B n'est pas nul, le compteur ordinal est décrémenté de 2 et l'instruction est réexécutée.

Chemin des données :



Durée : B = 0 : 4 cycles M ; 16 temps T ; 8 used @ 2 MHz.

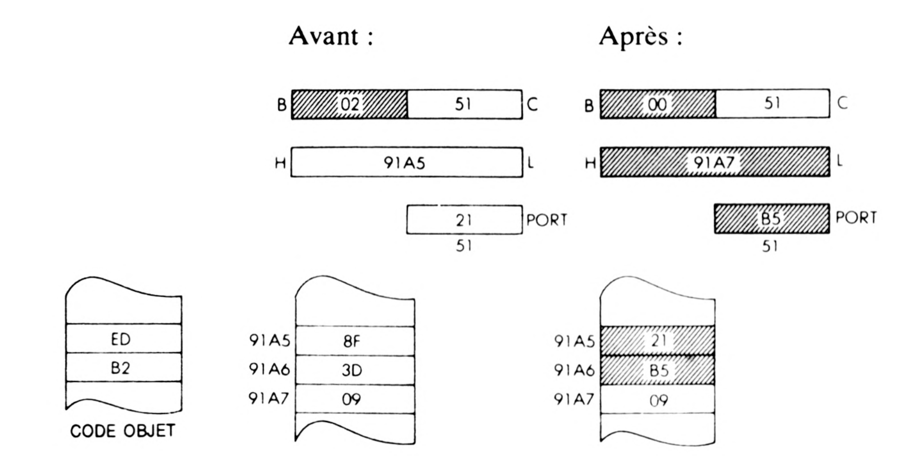
B ≠ 0 : 5 cycles M ; 21 temps T ; 10,5 usec @ 2 MHz.

Mode d'adressage : Externe

Indicateurs :

S	Z		H		P/V	N	C
?	1		?		?	1	

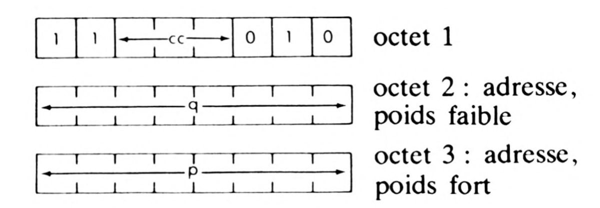
Exemple : INIR



JP cc, pq Saut conditionnel à l'adresse pq.

Fonction : si cc vraie : PC ← pq

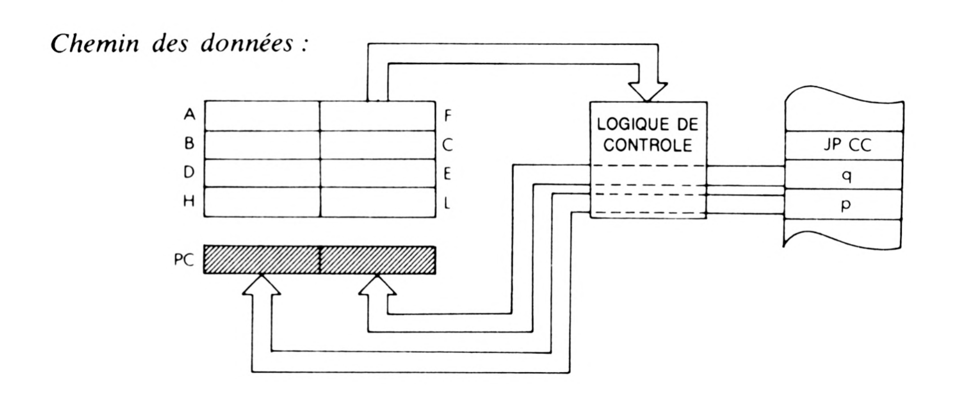
Format :



Description : Si la condition spécifiée est vraie, les deux octets d'adresse qui suivent immédiatement le code opératoire sont chargés dans le compteur ordinal, le premier octet suivant le code opératoire étant chargé dans les poids faibles du PC. Si la condition n'est pas remplie, l'adresse est ignorée. cc peut être n'importe lequel de :

NZ	– 000	non nul
Z	– 001	nul
NC	– 010	pas de report
C	– 011	report
PO	– 100	parité impaire
PE	– 101	parité paire
P	– 110	plus
M	– 111	moins

Chemin des données :



Durée : 3 cycles M ; 10 temps T ; 5 usec @ 2 MHz

Mode d'adressage : Immédiat.

Codes :

CC	NZ	Z	NC	C	PO	PE	P	M
	C2	CA	D2	DA	E2	EA	F2	FA

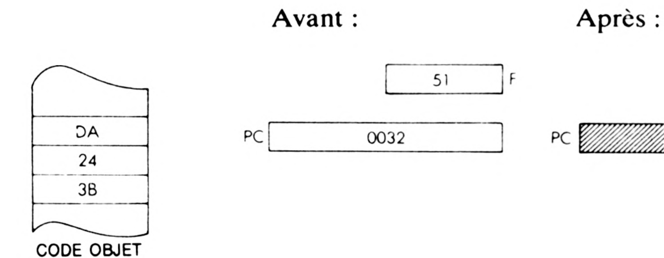
Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : JP C, 3B24

Avant : Après :



JP pq

Saut à l'adresse pq.

Fonction : PC ← pq

Format :

1	1	0	0	0	0	1	1

octet 1 : C3

← q →

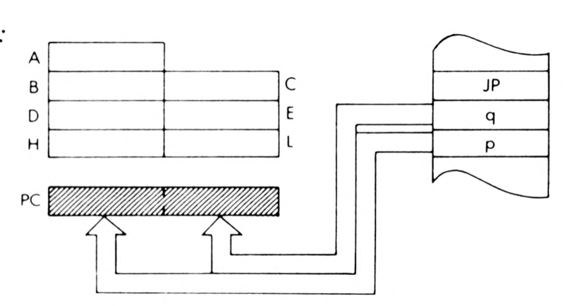
octet 2 : adresse, poids faible

← p →

octet 3 : adresse, poids fort

Description : Le contenu de l'emplacement mémoire suivant immédiatement le code opératoire est chargé dans la moitié basse du compteur ordinal et le contenu du second emplacement mémoire suivant immédiatement le code opératoire est chargé dans la moitié haute du compteur ordinal. L'instruction suivante est cherchée à cette nouvelle adresse.

Chemin des données :



Durée : 3 cycles M ; 10 temps T ; 5 usec @ 2 MHz

Mode d'adressage : Immédiat.

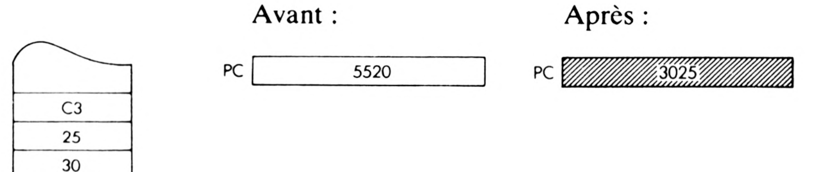
Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : JP 3025

Avant : Après :



JP (HL)

Saut au contenu de HL.

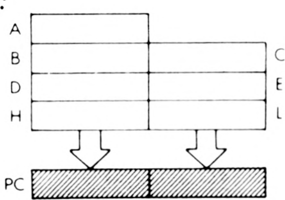
Fonction : PC ← HL

Format :

1	1	1	0	1	0	0	1	E9

Description : Le contenu du registre double HL est chargé dans le compteur ordinal. L'instruction suivante est cherchée à cette nouvelle adresse.

Chemin des données :



Durée : 1 cycle M ; 4 temps T ; 2 usec @ 2 MHz

Mode d'adressage : Implicite.

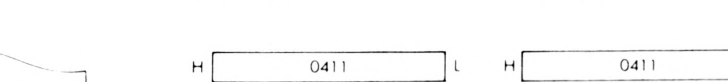
Indicateurs :

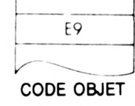
S	Z		H		P/V	N	C

(aucun effet)

Exemple : JP (HL)

Avant : Après :





JP (IX)

Saut au contenu de IX.

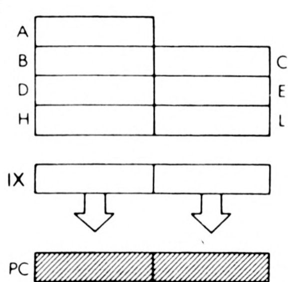
Fonction : PC ← IX

Format :

1	1	0	1	1	1	0	1	octet 1 : DD
1	1	1	0	1	0	0	1	octet 2 : E9

Description : Le contenu du registre IX est chargé dans le compteur ordinal. L'instruction suivante est cherchée à cette nouvelle adresse.

Chemin des données :



Durée : 2 cycles M ; 8 temps T ; 4 usec @ 2 MHz

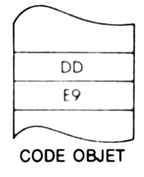
Mode d'adressage : Implicite

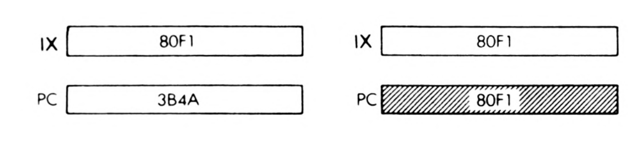
Indicateurs :

S	Z		H		P/V	N	C	
								(aucun effet)

Exemple : JP (IX)

Avant : Après :





JP (IY)

Saut au contenu de IY.

Fonction : PC ← IY

Format :

1	1	1	1	1	1	0	1	octet 1 : FD
1	1	1	0	1	0	0	1	octet 2 : E9

Description : Le contenu du registre IY est chargé dans le compteur ordinal. L'instruction suivante est cherchée à cette nouvelle adresse.

Chemin des données :

A B C D E H L IY PC

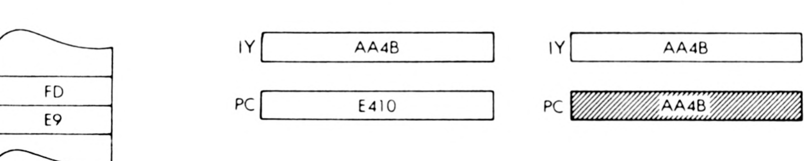
Durée : 2 cycles M ; 8 temps T ; 4 usec @ 2 MHz

Mode d'adressage : Implicite

Indicateurs : S Z H P/V N C (aucun effet)

Exemple : JP (IY)

Avant : **Après :**



JR cc, e Saut relatif de e conditionnel.

Fonction : si cc vrai, PC ← PC + e

Format :

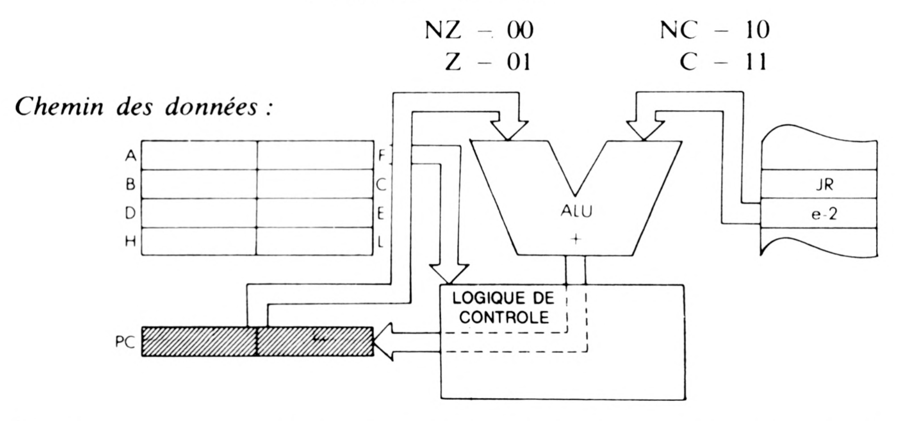
0	0	1	c	c	0	0	0	octet 1

← e-2 →	octet 2 : déplacement

Description : Si la condition spécifiée est remplie, le déplacement fourni est additionné au compteur ordinal, en utilisant l'arithmétique en complément à deux de façon à permettre à la fois des sauts en amont et en aval. Le déplacement est ajouté à la valeur de PC + 2 (après le saut). Ainsi le déplacement réel va de − 126 à + 129 octets. L'assembleur soustrait automatiquement 2 du déplacement source pour générer le code hexa. Si la condition n'est pas remplie, le déplacement est ignoré et l'exécution des instructions continue en séquence. cc peut être n'importe lequel de :

NZ – 00 NC – 10
Z – 01 C – 11

Chemin des données :



Durée :

	cycles M :	*temps T :*	*usec* @ 2 MHz:
condition remplie:	3	12	6
condition non remplie:	2	7	3.5

Mode d'adressage : Immédiat.

Codes :

cc:	NZ	Z	NC	C
	20	28	30	38

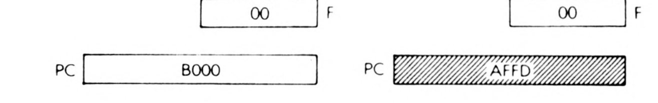
Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : JR NC, $ – 3 $ = PC courant

Avant : Après :



30
FB

CODE OBJET

JR e Saut relatif de e.

Fonction : PC ← PC + e

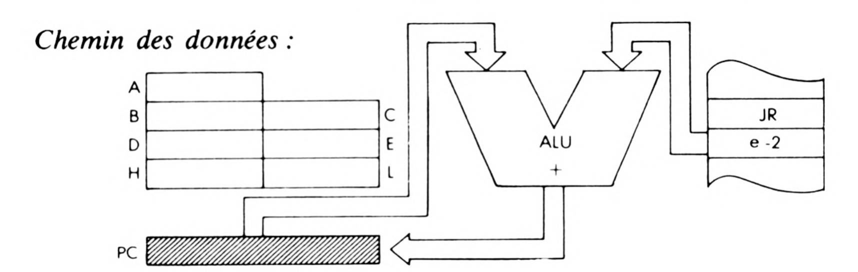
Format :

0	0	0	1	1	0	0	0	octet 1 : 18

← e-2 →	octet 2 : déplacement

Description : Le déplacement fourni est additionné au compteur ordinal en utilisant l'arithmétique en complément à deux de façon à permettre à la fois des sauts en amont et en aval. Le déplacement est ajouté à la valeur de PC + 2 (après le saut). Ainsi le déplacement réel va de − 126 à + 129 octets. L'assembleur soustrait automatiquement 2 du déplacement source pour générer le code hexa.

Chemin des données :



Durée : 3 cycles M ; 12 temps T ; 6 usec @ 2 MHz

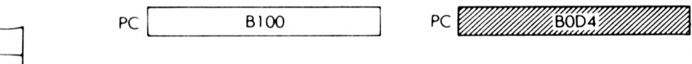
Mode d'adressage : Immédiat.

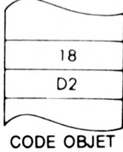
Indicateurs :

S	Z		H		P/V	N	C	
								(aucun effet)

Exemple : JR D4

Avant : Après :

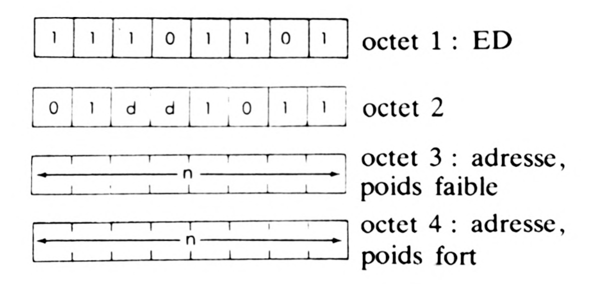




LD dd, (nn) Chargement du registre double dd à partir de l'emplacement mémoire d'adresse nn.

Fonction : $dd_{bas} \leftarrow (nn)$; $dd_{haut} \leftarrow (nn + 1)$

Format :

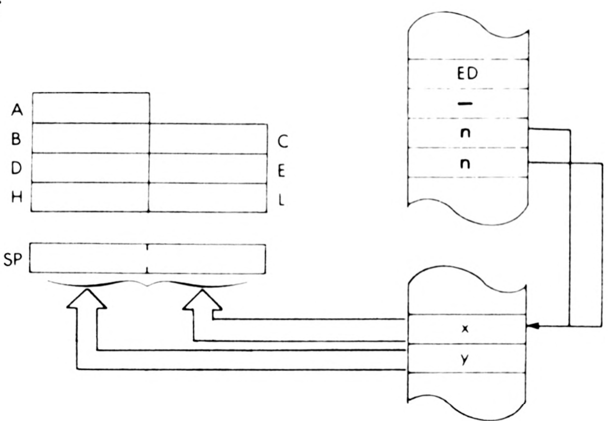


Description : Le contenu de l'emplacement mémoire adressé par les emplacements mémoires suivant immédiatement le code opératoire est chargé dans les poids faibles du registre double spécifié. Le contenu de l'emplacement mémoire suivant immédiatement celui chargé précédemment est ensuite chargé dans les poids forts du registre double. L'octet de poids faible de l'adresse nn suit immédiatement le code opératoire. dd peut être n'importe lequel de :

BC – 00	HL – 10
DE – 01	SP – 11



Chemin des données :



Durée : 6 cycles M ; 20 temps T ; 10 usec @ 2 MHz

Mode d'adressage : Direct.

Codes :

dd:	BC	DE	HL	SP
	4B	5B	6B	7B

Indicateurs :

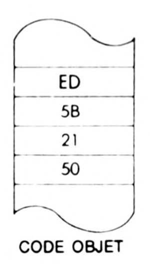
S	Z		H		P/V	N	C

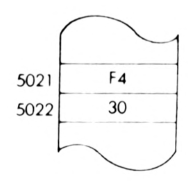
(aucun effet)

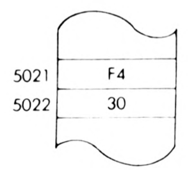
Exemple : LD DE, (5021)











LD dd, nn Chargement du registre double dd avec la donnée immédiate nn.

Fonction : dd ← nn

Format :

0	0	d	d	0	0	0	1

octet 1

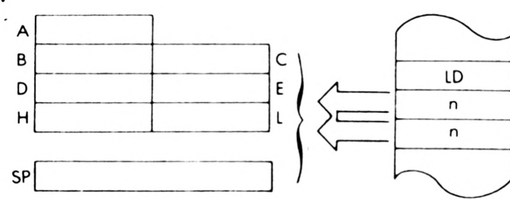
← n → octet 2 : donnée immédiate poids faible

← n → octet 3 : donnée immédiate poids fort

Description : Le contenu des deux emplacements mémoire suivant immédiatement le code opératoire est chargé dans le registre double spécifié. L'octet de poids faible de la donnée est situé juste derrière le code opératoire. dd peut être n'importe lequel de :

BC – 00	HL – 10
DE – 01	SP – 11

Chemin des données :



Durée : 3 cycles M ; 10 temps T ; 5 usec @ 2 MHz

Mode d'adressage : Immédiat.

Codes :

dd: BC	DE	HL	SP
01	11	21	31

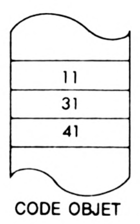
Indicateurs :

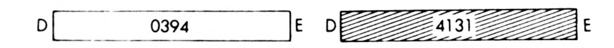
S	Z		H		P/V	N	C

(aucun effet)

Exemple : LD DE, 4131

Avant : Après :



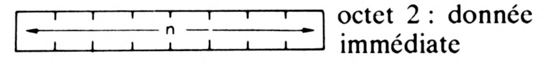


LD r, n

Chargement du registre r avec la donnée immédiate n.

Fonction : r ← n

Format :

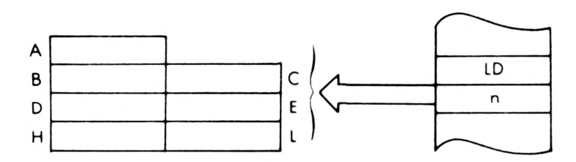


Description :

Le contenu de l'emplacement mémoire suivant immédiatement le code opératoire est chargé dans le registre spécifié. r peut être n'importe lequel de :

A – 111 E – 011
B – 000 H – 100
C – 001 L – 101
D – 010

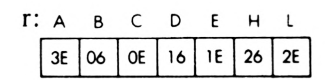
Chemin des données :



Durée : 2 cycles M ; 7 temps T ; 3,5 usec @ 2 MHz

Mode d'adressage : Immédiat.

Codes :



r: A	B	C	D	E	H	L
3E	06	0E	16	1E	26	2E

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : LD C, 3B

Avant : Après :

C 01 C 3B

0E
3B

CODE OBJET

LD r, r'

Chargement du registre r à partir du registre r'.

Fonction : r ← r'

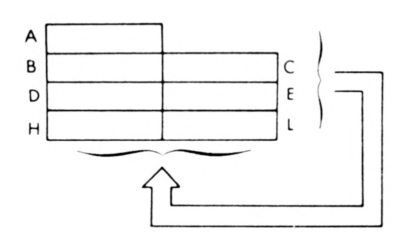
Format :

0	1	← r →	← r' →

Description : Le contenu du registre source spécifié est chargé dans le registre destination spécifié. r et r' peuvent être n'importe lesquels de :

A – 111
B – 000
C – 001
D – 010
E – 011
H – 100
L – 101

Chemin des données :



Durée : 1 cycle M ; 4 temps T ; 2 usec @ 2 MHz

Mode d'adressage : Implicite.

Codes :

(dest.) \ (source)	A	B	C	D	E	H	L
A	7F	78	79	7A	7B	7C	7D
B	47	40	41	42	43	44	45
C	4F	48	49	4A	4B	4C	4D
D	57	50	51	52	53	54	55
E	5F	58	59	5A	5B	5C	5D
H	67	60	61	62	63	64	65
L	6F	68	69	6A	6B	6C	6D

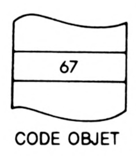
Indicateurs :

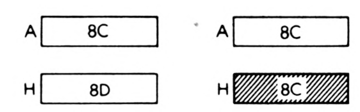
S	Z		H		P/V	N	C

(aucun effet)

Exemple : LD H, A

Avant : Après :





LD (BC), A

Chargement de l'emplacement mémoire d'adresse indirecte (BC) à partir de l'accumulateur.

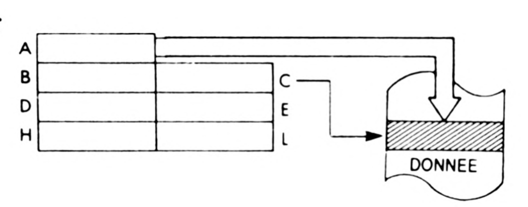
Fonction : (BC) ← A

Format :

0	0	0	0	0	0	1	0	02

Description : Le contenu de l'accumulateur est chargé dans l'emplacement mémoire adressé par le contenu du registre double BC

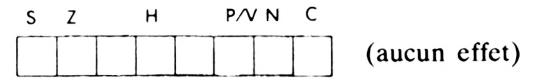
Chemin des données :



Durée : 2 cycles M ; 7 temps T ; 3,5 usec @ 2 MHz

Mode d'adressage : Indirect.

Indicateurs :

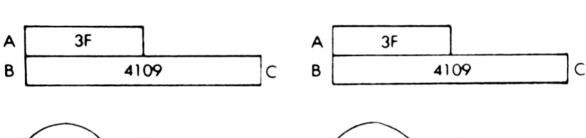


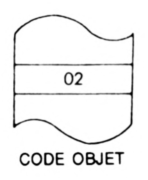
(aucun effet)

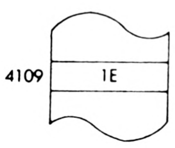
Exemple : LD (BC), A

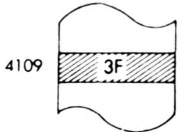


Avant : Après :









LD (DE), A

Chargement de l'emplacement mémoire d'adresse indirecte (DE) à partir de l'accumulateur.

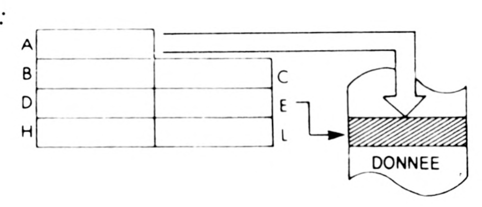
Fonction : (DE) ← A

Format :

0	0	0	1	0	0	1	0	12

Description : Le contenu de l'accumulateur est chargé dans l'emplacement mémoire adressé par le contenu du registre double DE

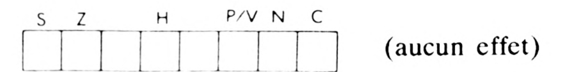
Chemin des données :



Durée : 2 cycles M ; 7 temps T ; 3,5 usec @ 2 MHz

Mode d'adressage : Indirect.

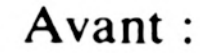
Indicateurs :



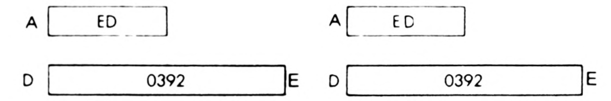
(aucun effet)

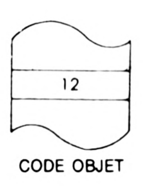
Exemple : LD (DE), A

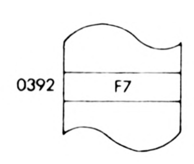
Avant : Après :

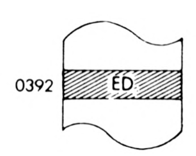








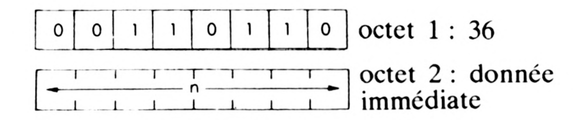




LD (HL), n — Chargement de la donnée immédiate n dans l'emplacement-mémoire d'adresse indirecte (HL).

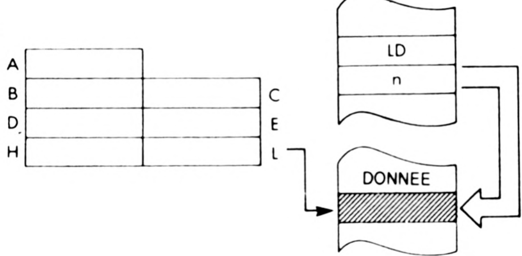
Fonction : (HL) ← n

Format :



Description : Le contenu de l'emplacement mémoire suivant immédiatement le code opératoire est chargé dans l'emplacement mémoire adressé indirectement par le pointeur de données HL


Chemin des données :



Durée : 3 cycles M ; 10 temps T ; 5 usec @ 2 MHz

Mode d'adressage : Immédiat/indirect.

Indicateur :

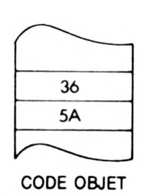
S	Z		H		P/V	N	C

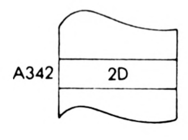
(aucun effet)

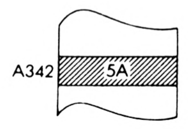
Exemple : LD (HL), 5A

Avant : Après :

H | A342 | L H | A342 | L







LD (HL), r Chargement de l'emplacement mémoire d'adresse indirecte (HL) à partir du registre r.

Fonction : (HL) ← r

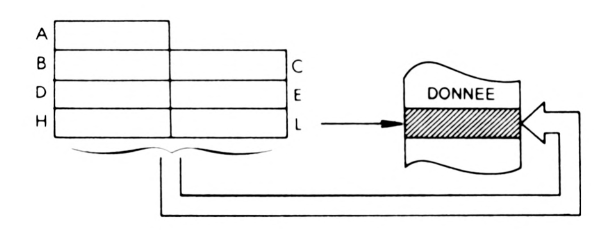
Format :

0	1	1	1	0	← r →

Description : Le contenu du registre spécifié est chargé dans l'emplacement mémoire adressé par le registre double HL. r peut être n'importe lequel de :

A – 111 E – 011
B – 000 H – 100
C – 001 L – 101
D – 010

Chemin des données :



Durée : 2 cycles M ; 7 temps T ; 3,5 usec @ 2 MHz

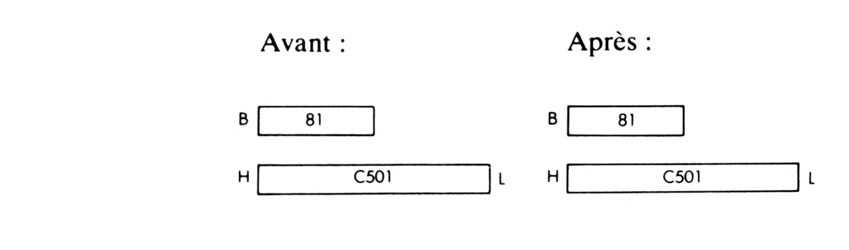
Mode d'adressage : Indirect.

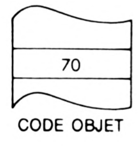
Codes :

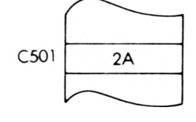
r:	A	B	C	D	E	H	L
	77	70	71	72	73	74	75

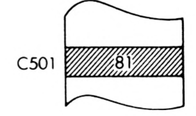
Indicateurs : S Z H P/V N C (aucun effet)

Exemple : LD (HL), B









LD r, (IX + d)

Chargement du registre r à partir de l'emplacement mémoire d'adresse indexée (IX+d).

Fonction : r ← (IX + d)

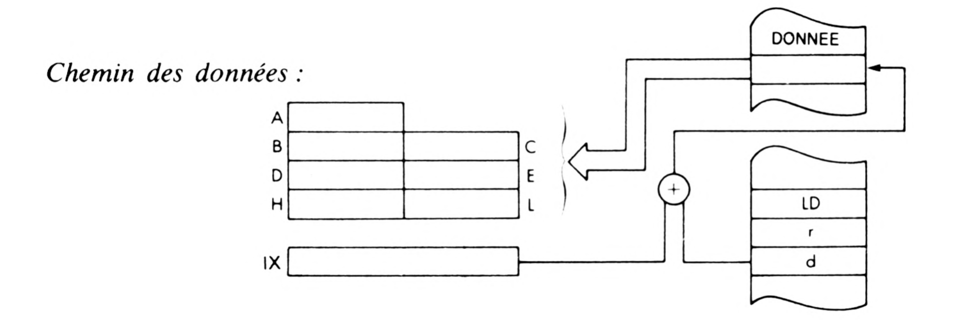
Format :

1	1	0	1	1	1	0	1	octet 1 : DD
0	1	←	r	→	1	1	0	octet 2
←			d				→	octet 3 : déplacement

Description : Le contenu de l'emplacement mémoire adressé par le registre d'index IX plus le déplacement fourni est chargé dans le registre spécifié. r peut être n'importe lequel de :

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Chemin des données :



Durée : 5 cycles M ; 19 temps T ; 9,5 usec @ 2 MHz

Mode d'adressage : Indexé.

Codes :

r:	A	B	C	D	E	H	L	
DD-	7E	46	4E	56	5E	66	6E	-d

Indicateurs : S Z H P/V N C (aucun effet)

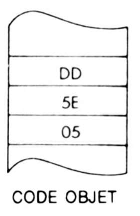
Exemple : LD E, (IX + 5)

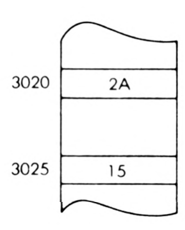
Avant : Après :

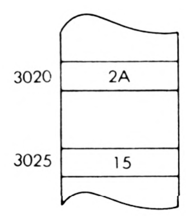


03 E — 15 E

IX 3020 — IX 3020







LD r, (IY + d)

Chargement du registre r à partir de l'emplacement mémoire d'adresse indexée (IY + d).

Fonction : r ← (IY + d)

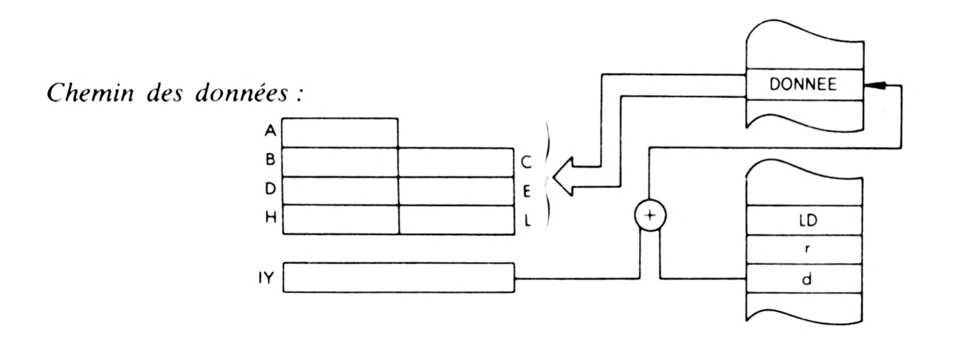
Format :

1	1	1	1	1	1	0	1	octet 1 : FD
0	1	←	r	→	1	1	0	octet 2
←			d				→	octet 3 : déplacement

Description : Le contenu de l'emplacement mémoire adressé par le registre d'index IY plus le déplacement fourni est chargé dans le registre spécifié. r peut être n'importe lequel de :

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Chemin des données :



Durée : 5 cycles M ; 19 temps T ; 9,5 usec @ 2 MHz

Mode d'adressage : Indexé.

Codes :

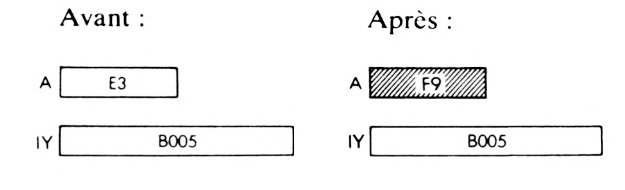
r:	A	B	C	D	E	H	L	
FD -	7E	46	4E	56	56	66	6E	- d

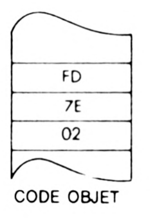
Indicateurs :

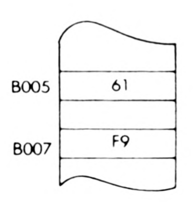
S	Z		H		P/V	N	C

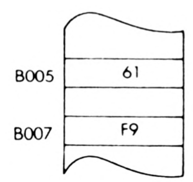
(aucun effet)

Exemple : LD A, (IY + 2)









LD (IX + d), n — Chargement de l'emplacement mémoire d'adresse indexée (IX + d) avec la donnée immédiate n.

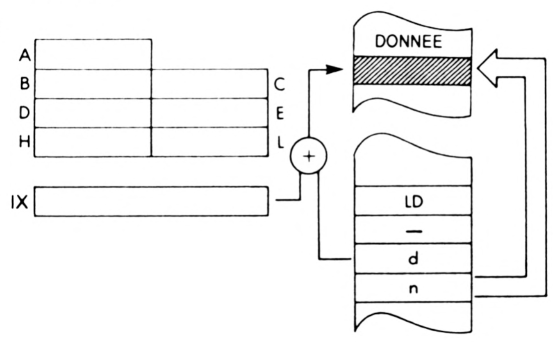
Fonction : (IX + d) ← n

Format :

1	1	0	1	1	1	0	1	octet 1 : DD
0	0	1	1	0	1	1	0	octet 2 : 36
←			d				→	octet 3 : déplacement
←			n				→	octet 4 : donnée immédiate

Description : Le contenu de l'emplacement mémoire suivant immédiatement le code opératoire est transféré dans l'emplacement mémoire adressé par le contenu du registre d'index IX plus le déplacement fourni.

Chemin des données :



Durée : 5 cycles M ; 19 temps T ; 9,5 usec @ 2 MHz

Mode d'adressage : Indexé/immédiat.

Indicateurs :

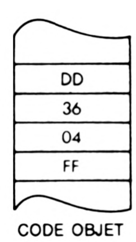
S	Z		H		P/V	N	C

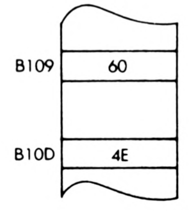
(aucun effet)

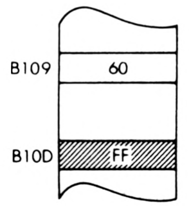
Exemple : LD (IX + 4), FF

Avant : IX B109

Après : IX B109







LD (IY + d), n Chargement de l'emplacement mémoire d'adresse indexée (IY + d) avec la donnée immédiate n.

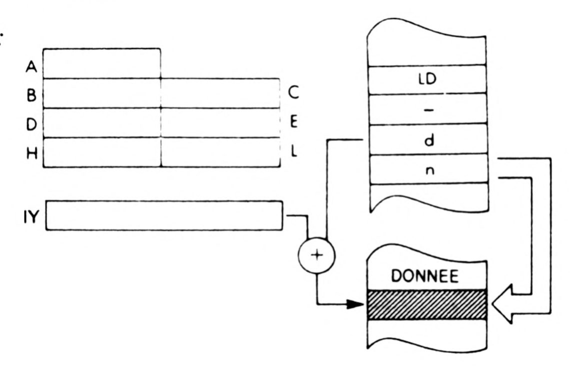
Fonction : (IY + d) ← n

Format :

1	1	1	1	1	1	0	1	octet 1 : FD
0	0	1	1	0	1	1	0	octet 2 : 36
←			d				→	octet 3 : déplacement
←			n				→	octet 4 : donnée immédiate

Description : Le contenu de l'emplacement mémoire suivant immédiatement le code opératoire est transféré dans l'emplacement mémoire adressé par le contenu du registre d'index IY plus le déplacement fourni.

Chemin des données :



Durée : 5 cycles M ; 19 Temps T ; 9,5 usec @ 2 MHz

Mode d'adressage : Indexé/immédiat.

Indicateurs :

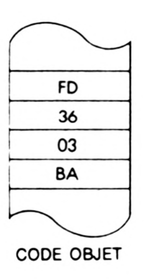
S	Z		H		P/V	N	C

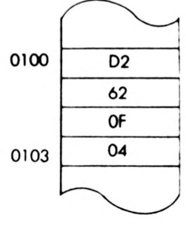
(aucun effet)

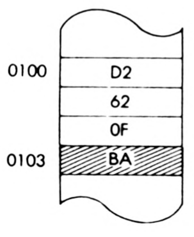
Exemple : LD (IY + 3), BA

Avant : Après :

IY 0100 IY 0100







LD (IX + d), r Chargement de l'emplacement mémoire d'adresse indexée (IX + d) à partir du registre r.

Fonction : (IX + d) ← r

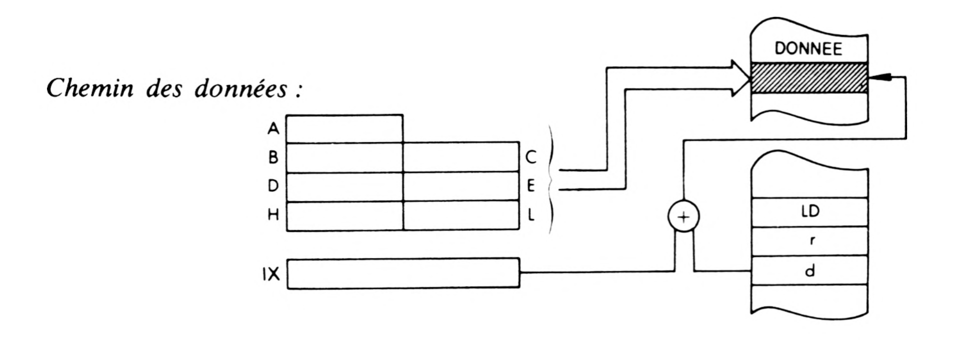
Format :

1	1	0	1	1	1	0	1	octet 1 : DD
0	1	1	1	0	←	r	→	octet 2
←			d				→	octet 3 : déplacement

Description : Le contenu du registre spécifié est chargé dans l'emplacement mémoire adressé par le contenu du registre d'index IX plus le déplacement fourni. r peut être n'importe lequel de :

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Chemin des données :



Durée : 5 cycles M ; 19 temps T ; 9,5 usec @ 2 MHz

Mode d'adressage : Indexé.

Codes :

r:	A	B	C	D	E	H	L	
DD-	77	70	71	72	73	74	75	- d

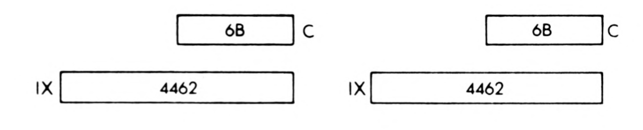
Indicateurs :

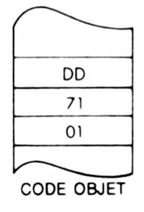
S	Z		H		P/V	N	C

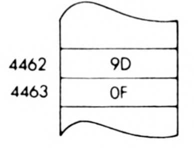
(aucun effet)

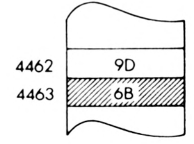
Exemple : LD (IX + 1), C

Avant : Après :







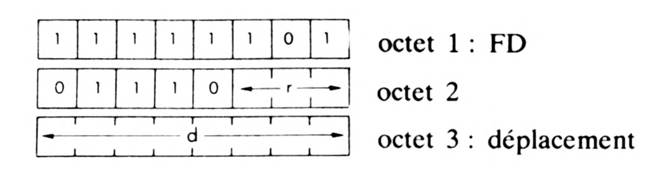


LD (IY + d), r

Chargement de l'emplacement mémoire d'adresse indexée (IY + d) à partir du registre r.

Fonction : (IY + d) ← r

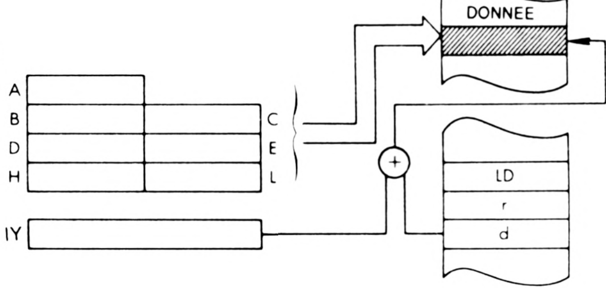
Format :



Description : Le contenu du registre spécifié est chargé dans l'emplacement mémoire adressé par le contenu du registre d'index IY plus le déplacement fourni. r peut être n'importe lequel de :

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	


Chemin des données :



Durée : 5 cycles M ; 19 temps T ; 9,5 usec @ 2 MHz

Mode d'adressage : Indexé.

Codes :

r:	A	B	C	D	E	H	L	
FD-	77	70	71	72	73	74	75	-d

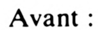


Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : **LD (IY + 3), A**



Avant :

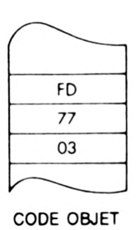
A: 3E

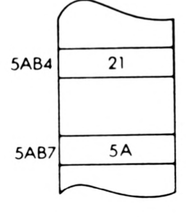
IY: 5AB4

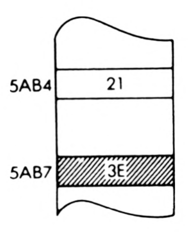
Après :

A: 3E

IY: 5AB4



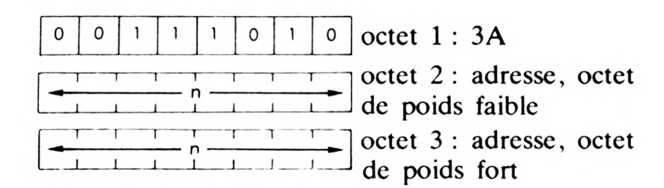




LD A, (nn) Chargement de l'accumulateur à partir de l'emplacement mémoire (nn).

Fonction : A ← (nn)

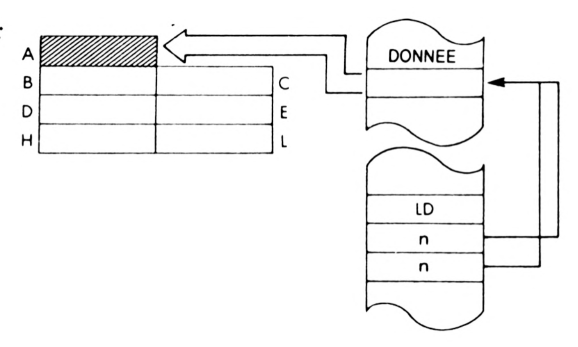
Format :



octet 1 : 3A
octet 2 : adresse, octet de poids faible
octet 3 : adresse, octet de poids fort

Description : Le contenu de l'emplacement mémoire adressé par le contenu des deux emplacements mémoire suivant immédiatemment le code opératoire est chargé dans l'accumulateur. L'octet de poids faible de l'adresse est situé juste derrière le code opératoire.

Chemin des données :

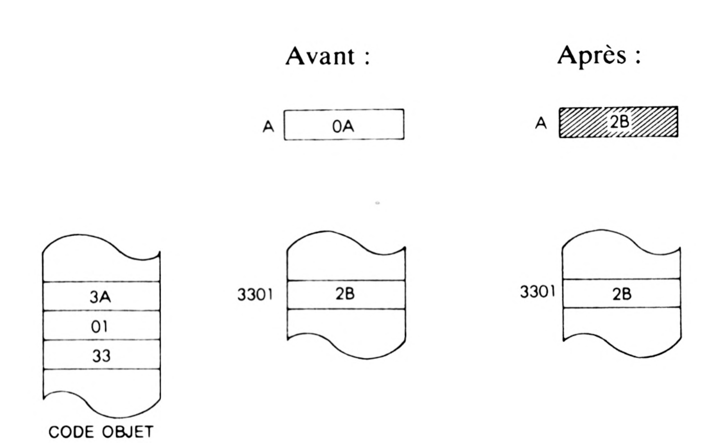


Durée : 4 cycles M ; 13 temps T ; 6,5 usec @ 2 MHz

Mode d'adressage : Direct.

Indicateurs : S Z H P/V N C (aucun effet)

Exemple : LD A, (3301)



LD (nn), A Chargement de l'emplacement mémoire d'adresse (nn) à partir de l'accumulateur.

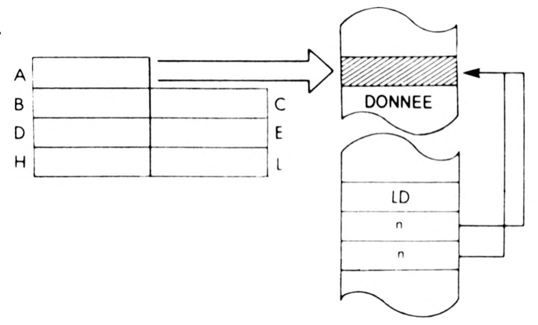
Fonction : (nn) ← A

Format :

0	0	1	1	0	0	1	0	octet 1 : 32
←			n				→	octet 2 : adresse, poids faible
←			n				→	octet 3 : adresse, poids fort

Description : Le contenu de l'accumulateur est chargé dans l'emplacement mémoire adressé par le contenu des 2 emplacements mémoire suivant immédiatement le code opératoire. L'octet de poids faible est situé juste derrière le code opératoire.

Chemin des données :



Durée : 4 cycles M ; 13 temps T ; 6,5 usec @ 2 MHz

Mode d'adressage : Direct.

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

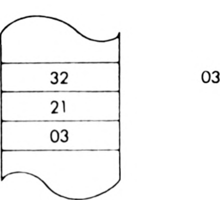
Exemple : LD (0321), A

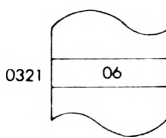
Avant :

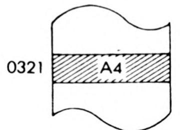
A A4

Après :

A A4



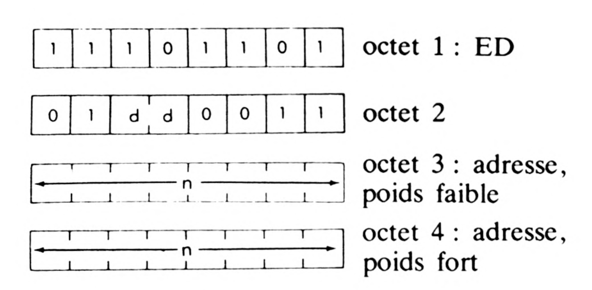




LD (nn), dd Chargement de l'emplacement mémoire d'adresse nn à partir du registre double dd.

Fonction : $(nn) \leftarrow dd_{bas}$; $(nn + 1) \leftarrow dd_{haut}$

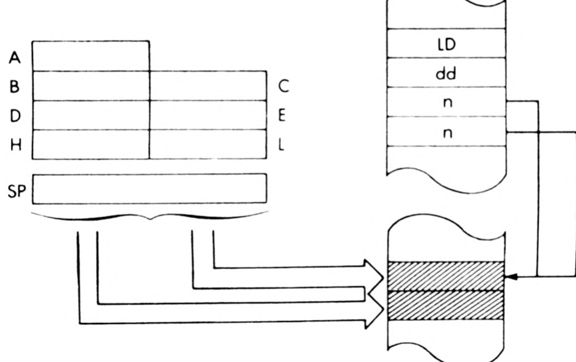
Format :



Descriptions : Le contenu de la partie basse du registre double spécifié est chargé dans l'emplacement mémoire adressé par le contenu des emplacements mémoire suivant immédiatement le code opératoire. Le contenu de la partie haute du registre double est chargé dans l'emplacement mémoire suivant immédiatement celui chargé à partir du poids faible. Le poids faible de l'adresse est situé juste derrière le code opératoire. dd peut être n'importe lequel de :

BC – 00	HL – 10
DE – 01	SP – 11


Chemin des données :



Durée : 6 cycles M ; 20 temps T ; 10 usec @ 2 MHz

Mode d'adressage : Direct.

Codes :

dd:	BC	DE	HL	SP
ED-	43	53	63	73

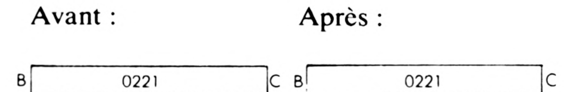
Indicateurs :

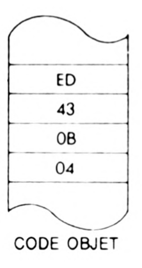
S	Z		H		P/V	N	C

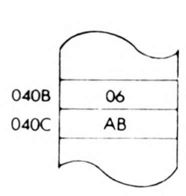
(aucun effet)

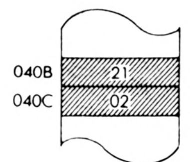
Exemple : LD (040B), BC

Avant : Après :









LD (nn), HL Chargement de l'emplacement mémoire d'adresse nn à partir de HL.

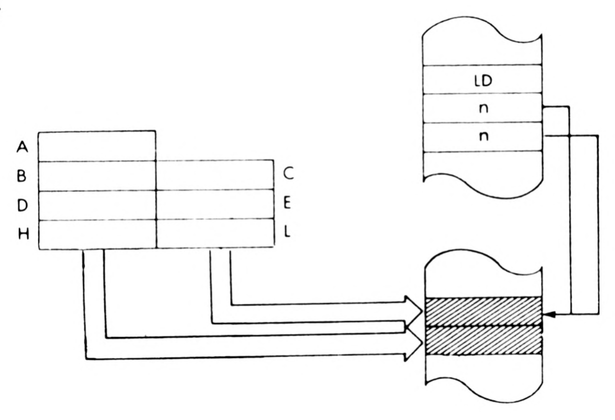
Fonction : (nn) ← L ; (nn + 1) ← H

Format :

0 0 1 0 0 0 1 0	octet 1 : 22
← n →	octet 2 : adresse, poids faible
← n →	octet 3 : adresse, poids fort

Description : Le contenu du registre L est chargé dans l'emplacement mémoire adressé par le contenu des emplacements mémoires suivant immédiatement le code opératoire. Le contenu du registre H est chargé dans l'emplacement mémoire suivant immédiatement celui chargé à partir du registre L. Le poids faible de l'adresse est situé juste derrière le code opératoire.

Chemin des données :



Durée : 5 cycles M ; 16 temps T ; 8 usec @ 2 MHz

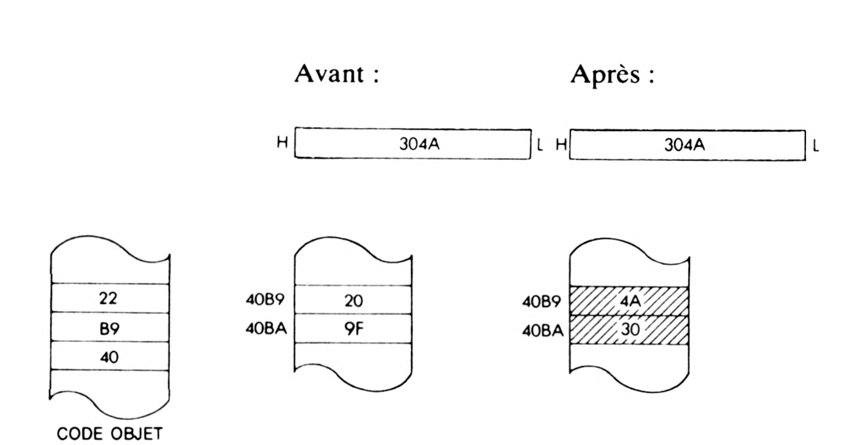
Mode d'adressage : Direct.

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

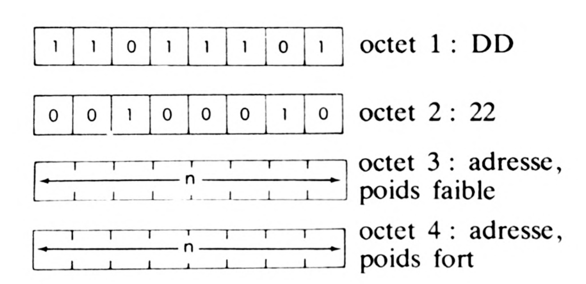
Exemple : LD (40B9), HL



LD (nn), IX — Chargement de l'emplacement mémoire d'adresse nn à partir de IX.

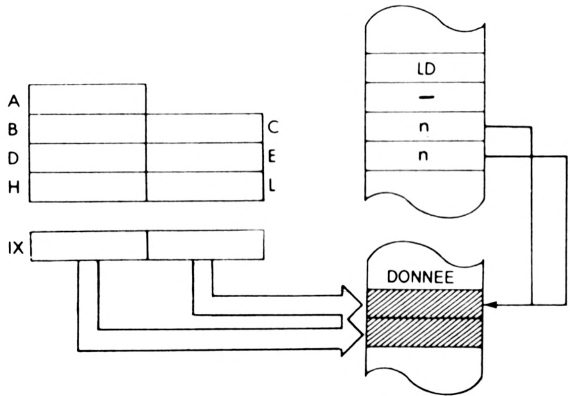
Fonction : $(nn) \leftarrow IX_{bas}$; $(nn + 1) \leftarrow IX_{haut}$

Format :



Description : Le contenu de la partie basse du registre IX est chargé dans l'emplacement mémoire adressé par le contenu des emplacements mémoire suivant immédiatement le code opératoire. Le contenu de la partie haute du registre IX est chargé dans l'emplacement mémoire suivant immédiatement celui chargé à partir de la partie basse. Le poids faible de l'adresse est situé juste derrière le code opératoire.


Chemin des données :



Durée : 6 cycles M ; 20 temps T ; 10 usec @ 2 MHz

Mode d'adressage : Direct.

Indicateurs :



(aucun effet)

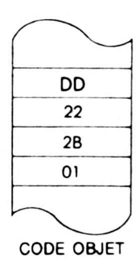
Exemple : LD (012B), IX

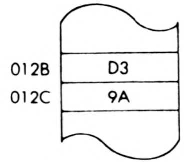
Avant :

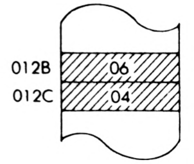
IX 0406

Après :

IX 0406



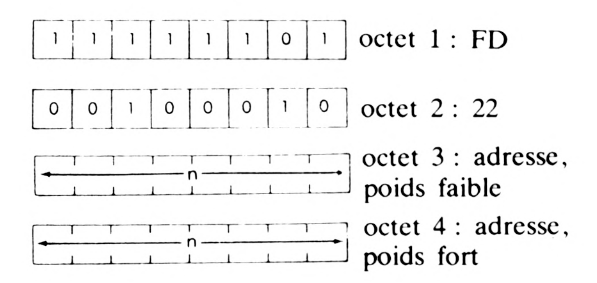




LD (nn), IY Chargement de l'emplacement mémoire d'adresse nn à partir de IY.

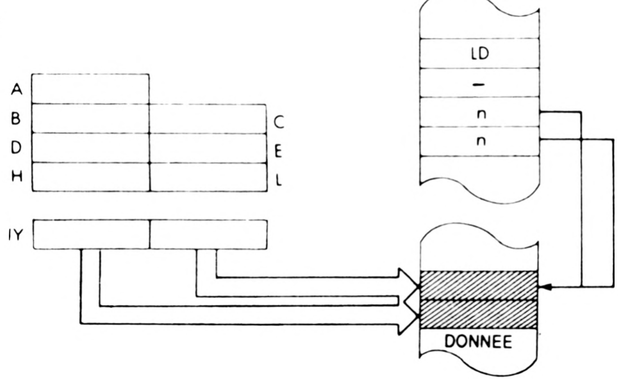
Fonction : $(nn) \leftarrow IY_{bas}$; $(nn + 1) \leftarrow IY_{haut}$

Format :



Description : Le contenu de la partie basse du registre IY est chargé dans l'emplacement mémoire adressé par le contenu des emplacements mémoire suivant immédiatement le code opératoire. Le contenu de la partie haute du registre IY est chargé dans l'emplacement mémoire suivant immédiatement celui chargé à partir de la partie basse. Le poids faible de l'adresse est situé juste derrière le code opératoire.


Chemin des données :



Durée : 6 cycles M ; 20 temps T ; 10 usec @ 2 MHz

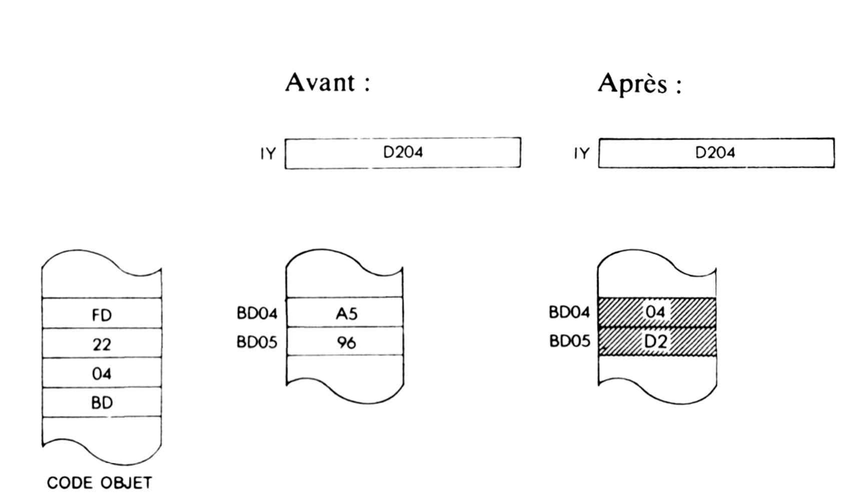
Mode d'adressage : Direct.

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : LD (BD04), IY



LD A, (BC) Chargement de l'accumulateur à partir de l'emplacement mémoire d'adresse indirecte (BC).

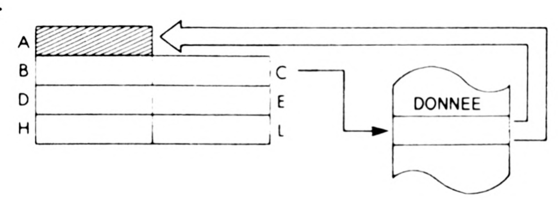
Fonction : A ← (BC)

Format :

0	0	0	0	1	0	1	0	0A

Description : Le contenu de l'emplacement mémoire adressé par le contenu du registre double BC est chargé dans l'accumulateur.

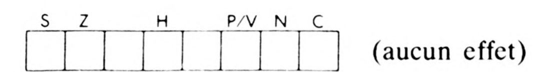
Chemin des données :



Durée : 2 cycles M ; 7 temps T ; 3,5 usec @ 2 MHz

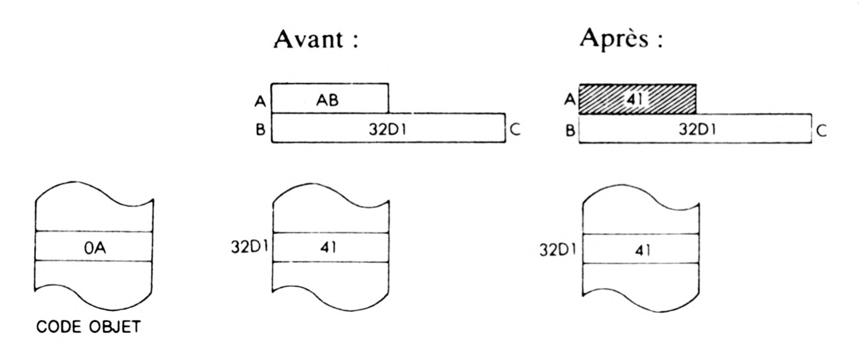
Mode d'adressage : Indirect.

Indicateurs :



(aucun effet)

Exemple : LD A, (BC)



LD A, (DE)

Chargement de l'accumulateur à partir de l'emplacement mémoire d'adresse indirecte (DE).

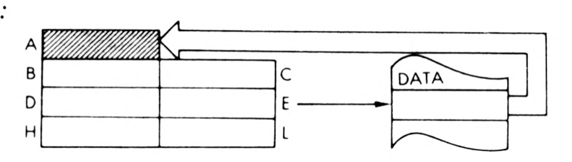
Fonction : A ← (DE)

Format :

0	0	0	1	1	0	1	0	1A

Description : Le contenu de l'emplacement mémoire adressé par le contenu du registre double DE est chargé dans l'accumulateur.

Chemin des données :



Durée : 2 cycles M ; 7 temps T ; 3,5 usec @ 2 MHz

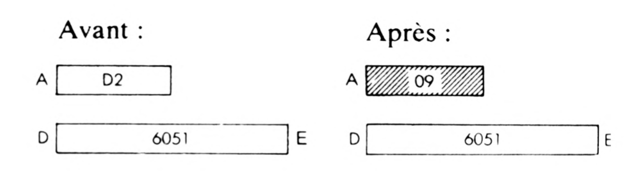
Mode d'adressage : Indirect.

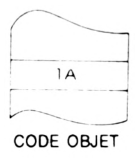
Indicateurs :

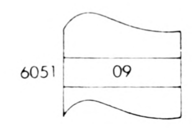
S	Z		H		P/V	N	C	
								(aucun effet)

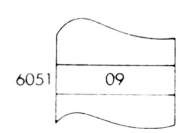
Exemple : LD A, (DE)

Avant : Après :









LD A, I

Chargement de l'accumulateur à partir du registre de vectorisation des interruptions I.

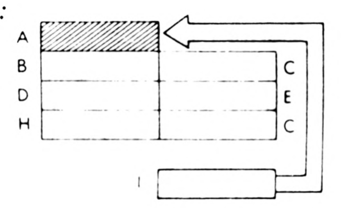
Fonction : A ← I

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
0	1	0	1	0	1	1	1	octet 2 : 57

Description : Le contenu du registre de vectorisation des interruptions est chargé dans l'accumulateur.

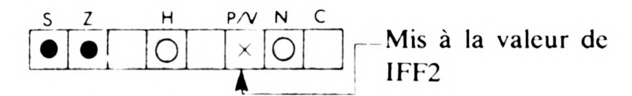
Chemin des données :



Durée : 2 cycles M ; 9 temps T ; 4,5 usec @ 2 MHz

Mode d'adressage : Implicite.

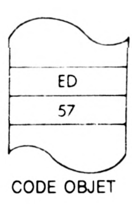
Indicateurs :

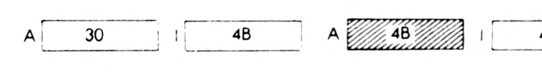


Mis à la valeur de IFF2

Exemple : LD A, I

Avant : Après :





LD I, A Chargement du registre de vectorisation des interruptions I à partir de l'accumulateur.

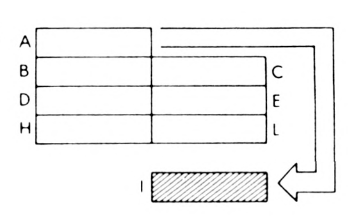
Fonction : I ← A

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
0	1	0	0	0	1	1	1	octet 2 : 47

Description : Le contenu de l'accumulateur est chargé dans le registre de vectorisation des interruptions.

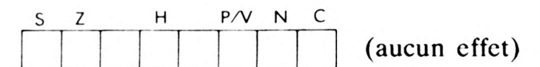
Chemin des données :



Durée : 2 cycles M ; 9 temps T ; 4,5 usec @ 2 MHz

Mode d'adressage : Implicite.

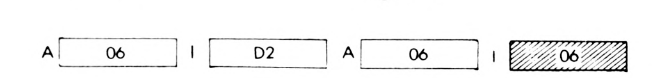
Indicateurs :

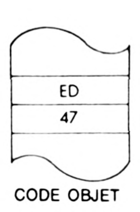


(aucun effet)

Exemple : LD I, A

Avant : Après :





LD A, R — Chargement de l'accumulateur à partir du registre de rafraîchissement mémoire R.

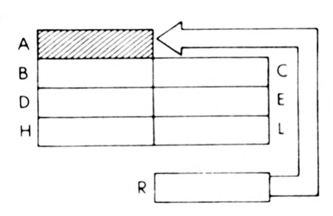
Fonction : A ← R

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
0	1	0	1	1	1	1	1	octet 2 : 5F

Description : Le contenu du registre de rafraîchissement mémoire est chargé dans l'accumulateur.

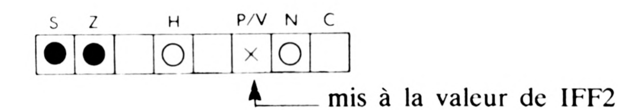
Chemin des données :



Durée : 2 cycles M ; 9 temps T ; 4,5 usec @ 2 MHz

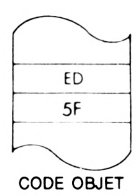
Mode d'adressage : Implicite.

Indicateurs :



Exemple : LD A, R

Avant : Après :

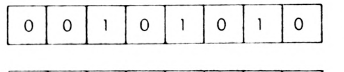




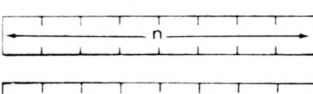
LD HL, (nn) Chargement du registre HL à partir de l'emplacement mémoire d'adresse nn.

Fonction : L ← (nn) ; H ← (nn + 1)

Format :



octet 1 : 2A

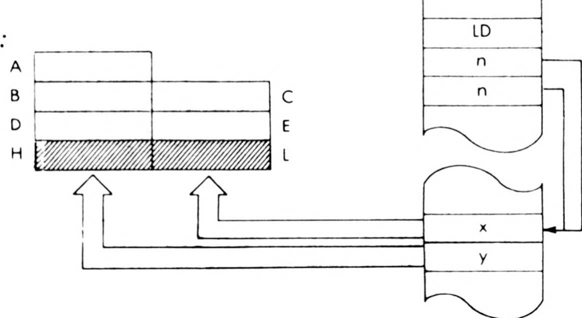


octet 2 : adresse, poids faible

octet 3 : adresse, poids fort

Description : Le contenu de l'emplacement mémoire adressé par le contenu des emplacements mémoire suivant immédiatement le code opératoire est chargé dans le registre L. Le contenu de l'emplacement mémoire suivant celui chargé dans le registre L est chargé dans le registre H. L'octet de poids faible de l'adresse nn est situé juste derrière le code opératoire.

Chemin des données :



Durée : 5 cycles M ; 16 temps T ; 8 usec @ 2 MHz

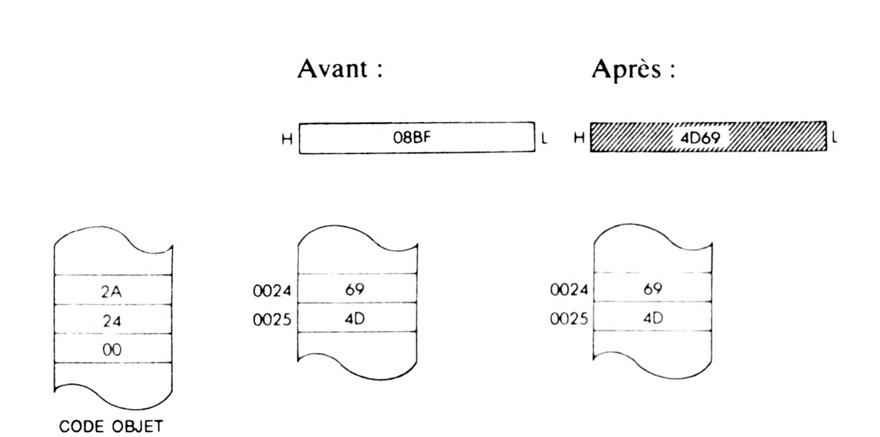
Mode d'adressage : Direct.

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : LD HL, (0024)



LD IX, nn

Chargement du registre IX avec la donnée immédiate nn.

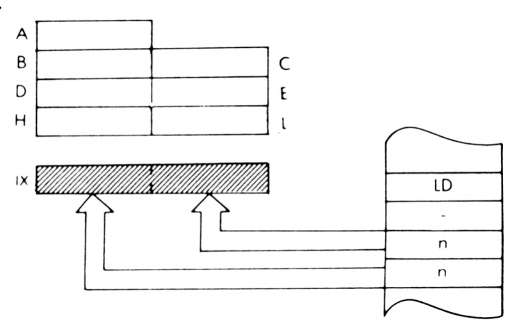
Fonction : IX ← nn

Format :

1	1	0	1	1	1	0	1	octet 1 : DD
0	0	1	0	0	0	0	1	octet 2 : 21
←			n				→	octet 3 : donnée immédiate, poids faible
←			n				→	octet 4 : donnée immédiate, poids fort

Description : Le contenu des emplacements mémoire suivant immédiatement le code opératoire est chargé dans le registre IX. L'octet de poids faible est situé juste derrière le code opératoire.

Chemin des données :



Durée : 4 cycles M ; 14 temps T ; 7 usec @ 2 MHz

Mode d'adressage : Immédiat.

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : LD IX, B0B1

Avant : Après :

IX 306F IX BCB1

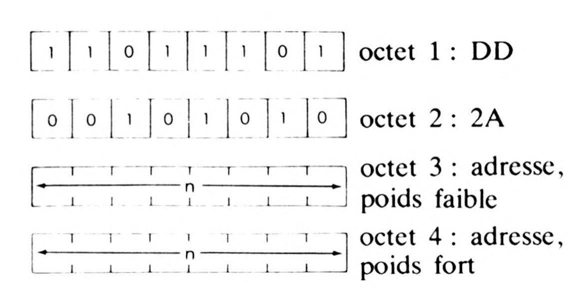
DD
21
B1
B0

CODE OBJET

LD IX, (nn) Chargement du registre IX à partir de l'emplacement mémoire d'adresse nn.

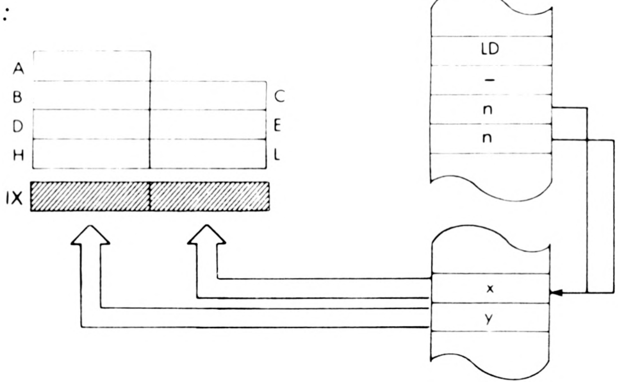
Fonction : $IX_{bas} \leftarrow (nn)$; $IX_{haut} \leftarrow (nn + 1)$

Format :



Descriptions : Le contenu de l'emplacement mémoire adressé par le contenu des emplacements mémoire suivant immédiatement le code opératoire est chargé dans la partie basse du registre IX. Le contenu de l'emplacement mémoire suivant immédiatement celui chargé dans la partie basse est chargé dans la partie haute du registre IX. Le poids faible de l'adresse nn est situé juste derrière le code opératoire.

Chemin des données :

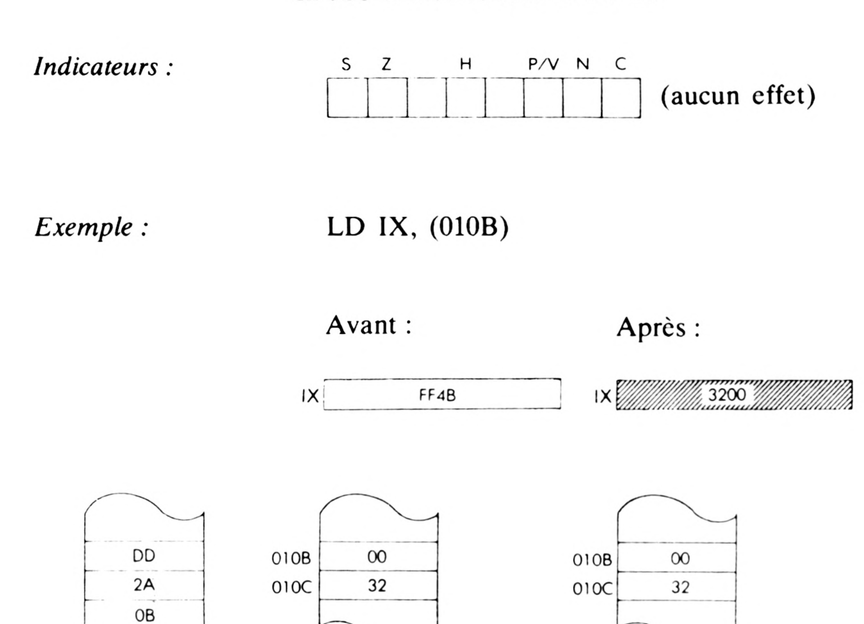


Durée : 6 cycles M ; 20 temps T ; 10 usec @ 2 MHz

Mode d'adressage : Direct.

Indicateurs : (aucun effet)

Exemple : LD IX, (010B)



LD IY, nn Chargement du registre IY avec la donnée immédiate nn.

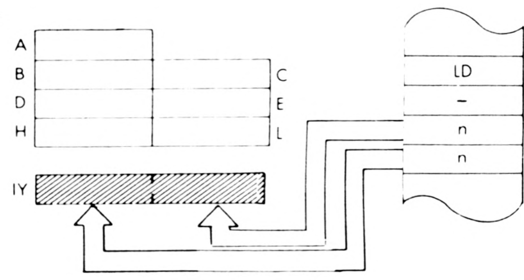
Fonction : IY ← nn

Format :

1	1	1	1	1	1	0	1	octet 1 : FD
0	0	1	0	0	0	0	1	octet 2 : 21
← n →								octet 3 : donnée immédiate, poids faible
← n →								octet 4 : donnée immédiate, poids fort

Description : Le contenu des emplacements mémoire suivant immédiatement le code opératoire est chargé dans le registre IY. L'octet de poids faible est situé juste derrière le code opératoire.

Chemin des données :



Durée : 4 cycles M ; 14 temps T ; 7 usec @ 2 MHz

Mode d'adressage : Immédiat.

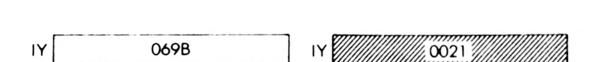
Indicateurs :

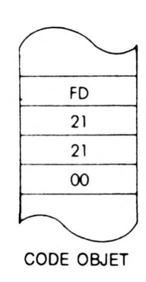
S	Z		H		P/V	N	C

(aucun effet)

Exemple : LD IY, 21

Avant : Après :

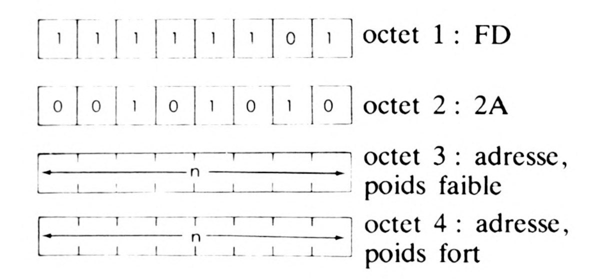




LD IY, (nn) Chargement du registre IY à partir de l'emplacement mémoire d'adresse nn.

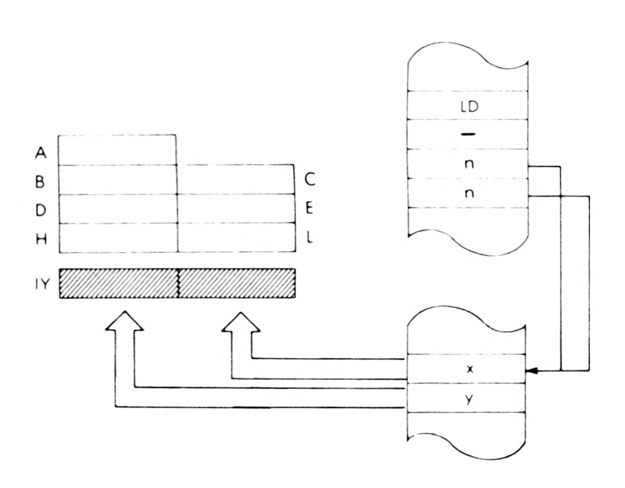
Fonction : $IY_{bas} \leftarrow (nn)$; $IY_{haut} \leftarrow (nn + 1)$

Format :



Description : Le contenu de l'emplacement mémoire adressé par le contenu des emplacements mémoire suivant immédiatement le code opératoire est chargé dans la partie basse du registre IY. Le contenu de l'emplacement mémoire suivant immédiatement celui chargé dans la partie basse est chargé dans la partie haute du registre IY. Le poids faible de l'adresse nn est situé juste derrière le code opératoire.

Chemin des données :



Durée : 6 cycles M ; 20 temps T ; 10 usec @ 2 MHz

Mode d'adressage : Direct.

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : LD IY, (500D)

Avant :

Après :

IY 6002

IY 4403

	FD
	2A
	0D
	50

CODE OBJET

Avant :

500D	03
500E	44

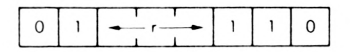
Après :

500D	03
500E	44

LD r, (HL) Chargement du registre r à partir de l'emplacement mémoire d'adresse indirecte (HL).

Fonction : r ← (HL)

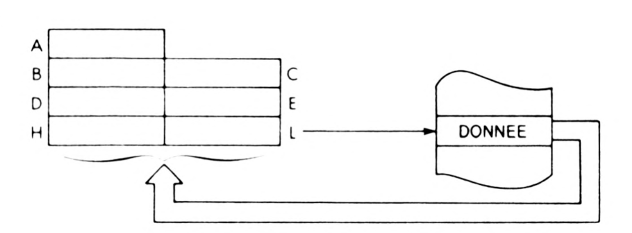
Format :



Description : Le contenu de l'emplacement mémoire adressé par HL est chargé dans le registre spécifié. r peut être n'importe lequel de :

A – 111
B – 000
C – 001
D – 010
E – 011
H – 100
L – 101

Chemin des données :



Durée : 2 cycles M ; 7 temps T ; 3,5 usec @ 2 MHz

Mode d'adressage : Indirect.

Codes :

r:	A	B	C	D	E	H	L
	7E	46	4E	56	5E	66	6E

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : LD D, (HL)

Avant :

D	3A		
H	0C	32	L

Après :

D	24		
H	0C	32	L

	56		

CODE OBJET

Avant : 0C32 : 24

Après : 0C32 : 24

LD R, A

Chargement du registre de rafraîchissement mémoire R à partir de l'accumulateur.

Fonction : R ← A

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
0	1	0	0	1	1	1	1	octet 2 : 4F

Description : Le contenu de l'accumulateur est chargé dans le registre de rafraîchissement mémoire.

Chemin des données :

A B C D E H L R

Durée : 2 cycles M ; 9 temps T ; 4,5 usec @ 2 MHz

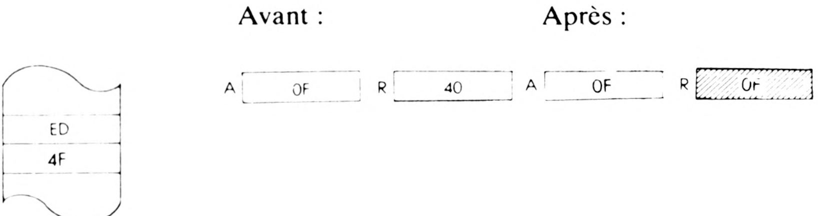
Mode d'adressage : Implicite.

Indicateurs :

S	Z		H		P/V	N	C	
								(aucun effet)

Exemple : LD R, A

Avant : Après :



LD SP, HL

Chargement du pointeur de pile à partir de HL.

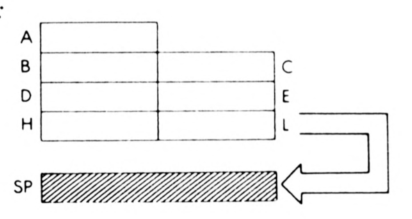
Fonction : SP ← HL

Format :

1	1	1	1	1	0	0	1	F9

Description : Le contenu du registre double HL est chargé dans le pointeur de pile.

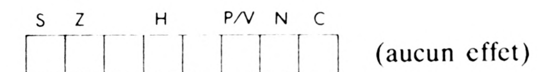
Chemin des données :



Durée : 1 cycle M ; 6 temps T ; 3 usec @ 2 MHz

Mode d'adressage : Implicite.

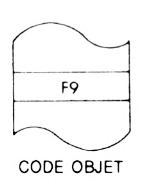
Indicateurs :

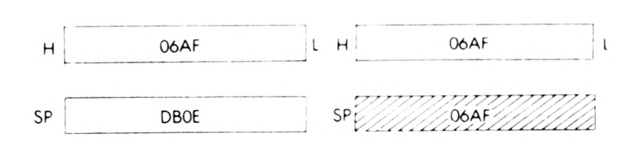


(aucun effet)

Exemple : LD SP, HL







LD SP, IX

Chargement du pointeur de pile à partir du registre IX.

Fonction : SP ← IX

Format :

1	1	0	1	1	1	0	1	octet 1 : DD
1	1	1	1	1	0	0	1	octet 2 : F9

Description : Le contenu du registre IX est chargé dans le pointeur de pile.

Chemin des données :

A B C D E H L IX SP

Durée : 2 cycles M ; 10 temps T ; 5 usec @ 2 MHz

Mode d'adressage : Implicite.

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : LD SP, IX

Avant :

IX	09D2
SP	54A0

Après :

IX	09D2
SP	09D2

DD
F9

CODE OBJET

LD SP, IY Chargement du pointeur de pile à partir du registre IY.

Fonction : SP ← IY

Format :

1	1	1	1	1	1	0	1	octet 1 : FD
1	1	1	1	1	0	0	1	octet 2 : F9

Description : Le contenu du registre IY est chargé dans le pointeur de pile.

Chemin des données :

A B C D E H L IY SP

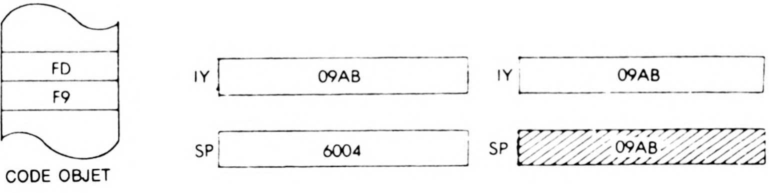
Durée : 2 cycles M ; 10 temps T ; 5 usec @ 2 MHz

Mode d'adressage : Implicite.

Indicateurs : S Z H P/V N C (aucun effet)

Exemple : LD SP, IY

Avant : Après :



LDD Transfert de bloc avec décrémentation.

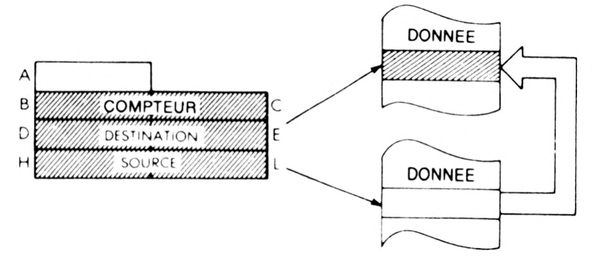
Fonction : (DE) ← (HL) ; DE ← DE − 1 ; HL ← HL − 1 ; BC ← BC − 1

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
1	0	1	0	1	0	0	0	octet 2 : A8

Description : Le contenu de l'emplacement mémoire adressé par HL est chargé dans l'emplacement mémoire adressé par DE. Ensuite BC, DE et HL sont tous trois décrémentés.

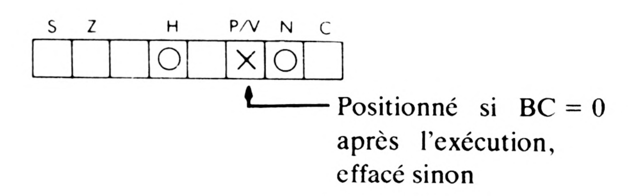
Chemin des données :



Durée : 4 cycles M ; 16 temps T ; 8 usec @ 2 MHz

Mode d'adressage : Indirect.

Indicateurs :



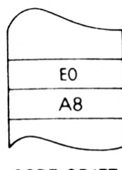
Exemple : LDD

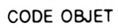
Avant :

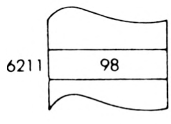
B	0B04	C
D	6211	E
H	843B	L

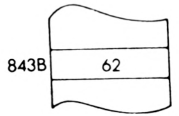
Après :

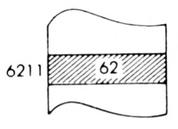


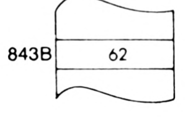












LDDR Transfert répétitif de bloc avec décrémentation.

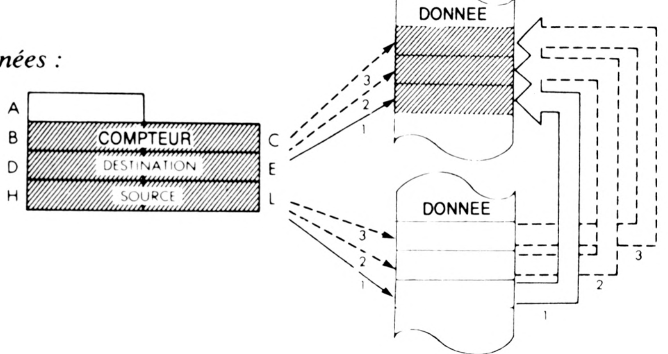
Fonction : (DE) ← (HL) ; DE ← DE − 1 ; HL ← HL − 1 ; BC ← BC − 1 ; Répéter jusqu'à BC = 0

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
1	0	1	1	1	0	0	0	octet 2 : B8

Description : Le contenu de l'emplacement mémoire adressé par HL est chargé dans l'emplacement mémoire adressé par DE. Ensuite DE, HL et BC sont tous trois décrémentés. Si BC ≠ 0, le compteur ordinal est décrémenté de 2 et l'instruction est réexécutée.


Chemin des données :



Durée : BC ≠ 0 : 5 cycles M ; 21 temps T ; 10,5 usec @ 2 MHz.

BC = 0 : 4 cycles M ; 16 temps T ; 8 usec @ 2 MHz.

Mode d'adressage : Indirect.

Indicateurs :

S	Z		H		P/V	N	C
			○		○	○	

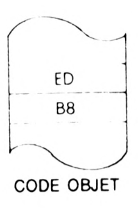
Exemple : LDDR

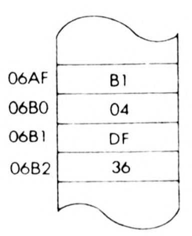
Avant :

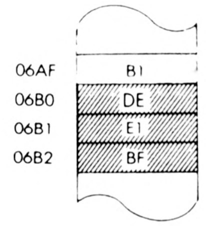
B	0003	C
D	06B2	E
H	9035	L

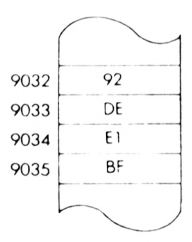
Après :

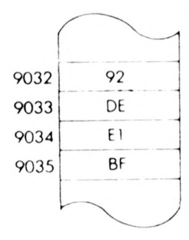
B	0000	C
D	06AF	E
H	9032	L











LDI

Transfert de bloc avec incrémentation.

Fonction :

(DE) ← (HL) ; DE ← DE + 1 ; HL ← HL + 1 ; BC ← BC − 1

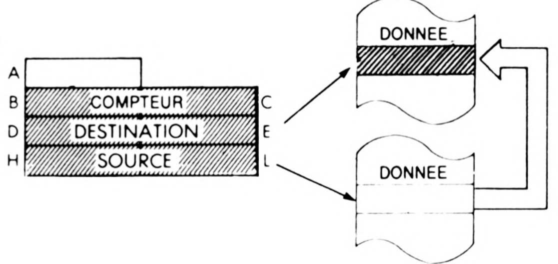
Format :

1	1	1	0	1	1	0	1	octet 1 : ED
1	0	1	0	0	0	0	0	octet 2 : AO

Description :

Le contenu de l'emplacement mémoire adressé par HL est chargé dans l'emplacement mémoire adressé par DE. Ensuite DE et HL sont tous deux incrémentés et le registre double BC est décrémenté.

Chemin des données :



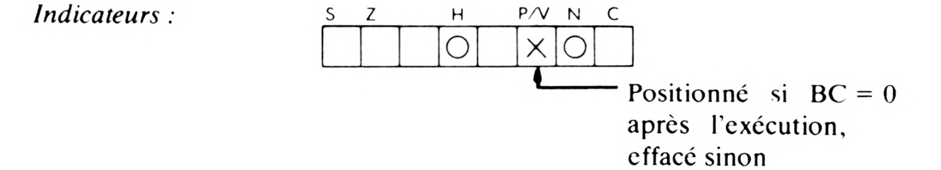
Durée :

4 cycles M ; 16 temps T ; 8 usec @ 2 MHz

Mode d'adressage :

Indirect.

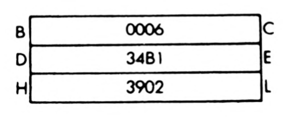
Indicateurs :

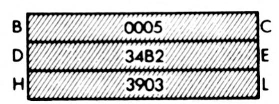


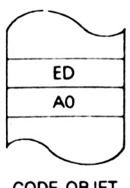
Exemple : LDI

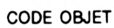
Avant :

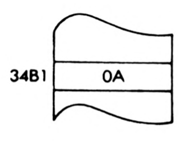
Après :

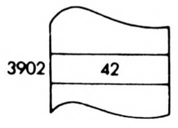


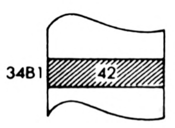


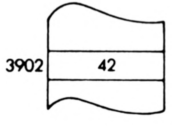












LDIR

Transfert répétitif de bloc avec incrémentation.

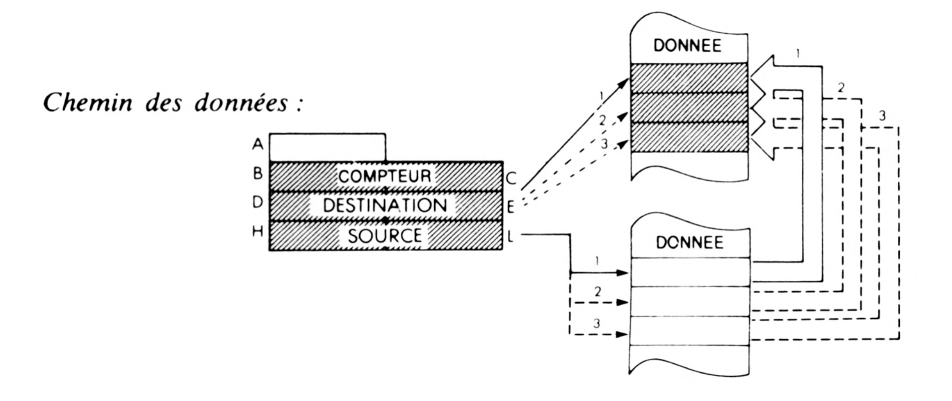
Fonction : (DE) ← (HL) ; DE ← DE + 1 ; HL ← HL + 1 ; BC ← BC − 1 ; Répéter jusqu'à BC = 0

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
1	0	1	1	0	0	0	0	octet 2 : BO

Description : Le contenu de l'emplacement mémoire adressé par HL est chargé dans l'emplacement mémoire adressé par DE. Ensuite DE et HL sont tous deux incrémentés. BC est décrémenté. Si BC ≠ 0 le compteur ordinal est décrémenté de 2 et l'instruction est réexécutée.

Chemin des données :



Durée : BC ≠ 0 : 5 cycles M ; 21 temps T ; 10,5 usec @ 2 MHz.

BC = 0 : 4 cycles M ; 16 temps T ; 8 usec @ 2 MHz.

Mode d'adressage : Indirect.

Indicateurs :

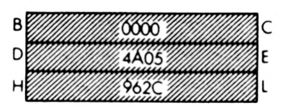
S	Z		H		P/V	N	C
			O		O	O	

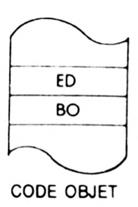
Exemple : LDIR

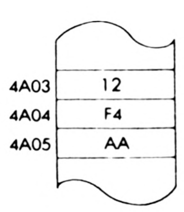
Avant :

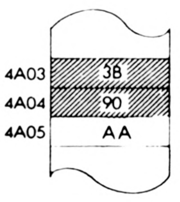
B	0002	C
D	4A03	E
H	962A	L

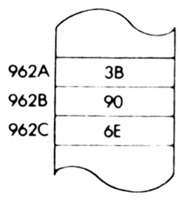
Après :

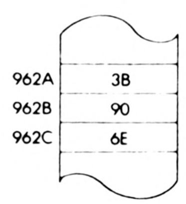












NEG

Opposé de l'accumulateur.

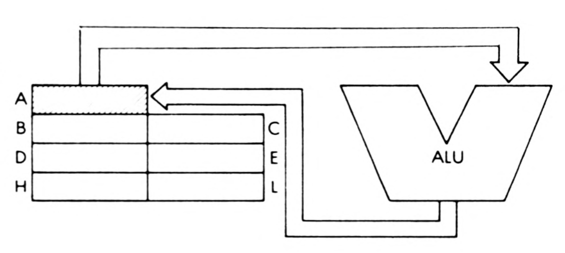
Fonction : A ← 0 − A

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
0	1	0	0	0	1	0	0	octet 2 : 44

Description : Le contenu de l'accumulateur est soustrait de zéro (complément à deux) et le résultat est rangé à nouveau dans l'accumulateur.

Chemin des données :



Durée : 2 cycles M ; 8 temps T ; 4 usec @ 2 MHz

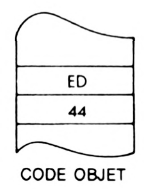
Mode d'adressage : Implicite.

Indicateurs :

S	Z		H		P/V	N	C
●	●		●		●	1	●

C est positionné si A valait 0 avant l'instruction.
P sera positionné si A valait 80H.

Exemple : NEG



Avant :

A 32

Après :

A CE

NOP Opération nulle.

Fonction : Délai

Format :

0	0	0	0	0	0	0	0	00

Description : Rien n'est fait pendant 1 cycle M.

Chemin des données : Aucune action

A			
B			C
D			E
H			L

Durée : 1 cycle M ; 4 temps T ; 2 usec @ 2 MHz

Mode d'adressage : Implicite.

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

OR s

Ou logique entre l'accumulateur et l'opérande s.

Fonction : A ← A ∨ s

Format : s : peut être r, n, (HL), (IX + d), ou (IY + d)

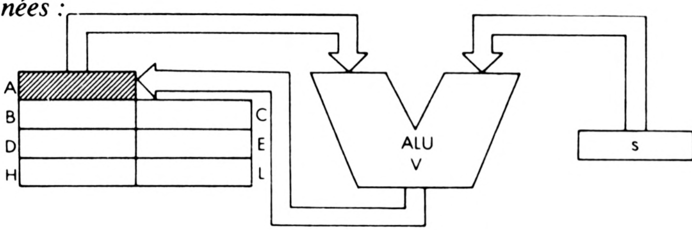
r	1	0	1	1	0	← r →			
n	1	1	1	1	0	1	1	0	octet 1 : F6
	← n →								octet 2 : donnée immédiate
(HL)	1	0	1	1	0	1	1	0	octet 1 : B6
(IX + d)	1	1	0	1	1	1	0	1	octet 1 : DD
	1	0	1	1	0	1	1	0	octet 2 : B6
	← d →								octet 3 : déplacement
(IY + d)	1	1	1	1	1	1	0	1	octet 1 : FD
	1	0	1	1	0	1	1	0	octet 2 : B6
	← d →								octet 3 : déplacement

r peut être n'importe lequel de :

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Description : Le ou logique de l'accumulateur et de l'opérande s spécifié est effectué et le résultat est rangé dans l'accumulateur. s est défini dans la description des instructions ADD similaires.

Chemin des données :

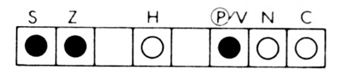


Durée :

s:	*cycles M :*	*temps T :*	*usec @ 2 MHz:*
r	1	4	4
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Mode d'adressage : r : implicite ; n : immédiat ; (HL) : indirect ; (IX + d), (IY + d) : indexé.

Codes : OR r



Indicateurs :

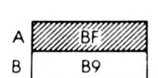
S	Z		H		P/V	N	C
●	●		○		●	○	○

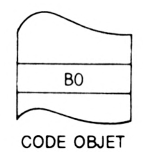
Exemple : OR B

Avant :

A	06
B	B9

Après :





OTDR Sortie par bloc avec décrémentation.

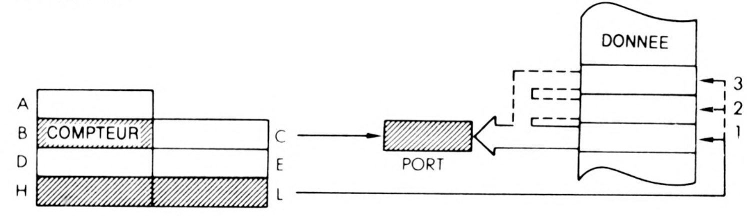
Fonction : (C) ← (HL) ; B ← B − 1 ; HL ← HL − 1 ; Répéter jusqu'à B = 0.

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
1	0	1	1	1	0	1	1	octet 2 : BB

Description : Le contenu de l'emplacement mémoire adressé par le registre double HL est écrit dans l'organe périphérique adressé par le contenu du registre C. Le registre B et le registre double HL sont tous deux décrémentés. Si B ≠ 0, le compteur ordinal est décrémenté de 2 et l'instruction est réexécutée. C fournit les bits A0 à A7 du bus adresse et B fournit (après décrémentation) les bits A8 à A15.

Chemin des données :



Durée : B = 0 : 4 cycles M ; 16 temps T ; 8 usec @ 2 MHz.

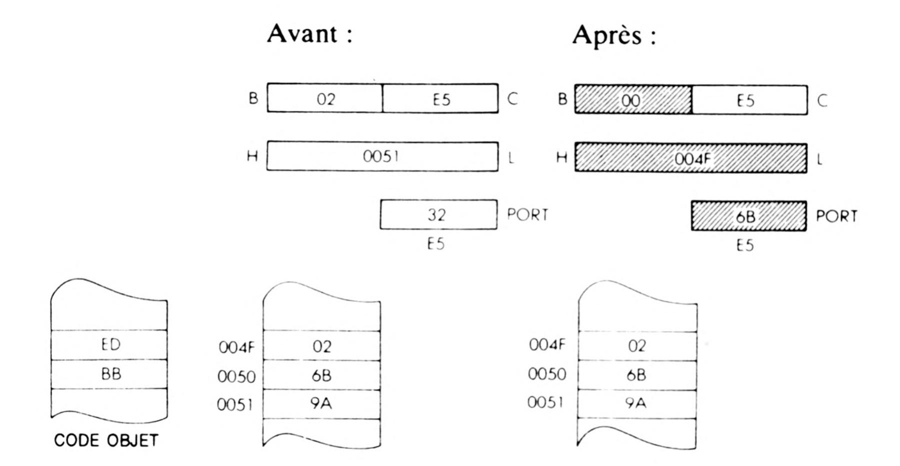
B ≠ 0 : 5 cycles M ; 21 temps T ; 10,5 usec @ 2 MHz

Mode d'adressage : Externe.

Indicateurs :

S	Z		H		P/V	N	C
?	1		?		?	1	

Exemple : OTDR



OTIR Sortie par bloc avec incrémentation.

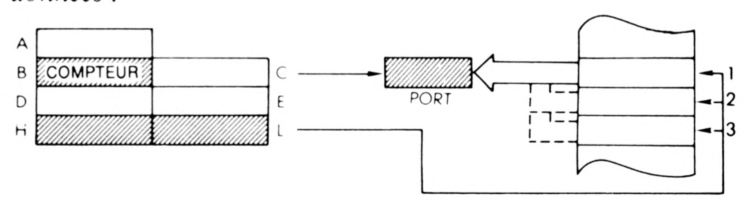
Fonction : (C) ← (HL) ; B ← B − 1 ; HL ← HL + 1 ; Répéter jusqu'à B = 0

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
1	0	1	1	0	0	1	1	octet 2 : B3

Description : Le contenu de l'emplacement mémoire adressé par le registre double HL est écrit dans l'organe périphérique adressé par le contenu du registre C. Le registre B est décrémenté et le registre double HL est incrémenté. Si B ≠ 0, le compteur ordinal est décrémenté de 2 et l'instruction est réexécutée. C fournit les bits A0 à A7 du bus adresse. B fournit (après décrémentation) les bits A8 à A15.

Chemin des données :



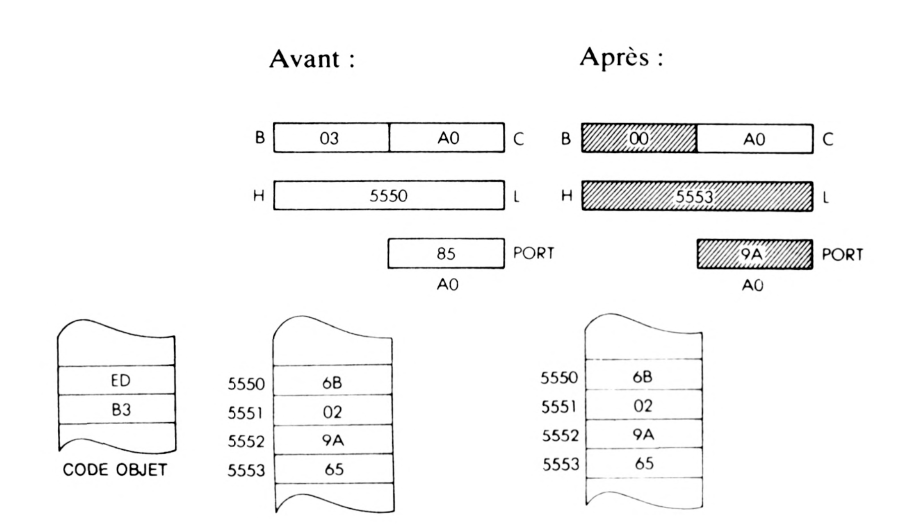
Durée : B = 0 : 4 cycles M ; 16 temps T ; 8 usec @ 2 MHz.
B ≠ 0 : 5 cycles M ; 21 temps T ; 10,5 usec @ 2 MHz

Mode d'adressage : Externe.

Indicateurs :

S	Z		H		P/V	N	C
?	1		?		?	1	

Exemple : OTIR



OUT (C), r

Sortie du registre r vers le port C.

Fonction : (C) ← r

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
0	1	←	r	→	0	0	1	octet 2

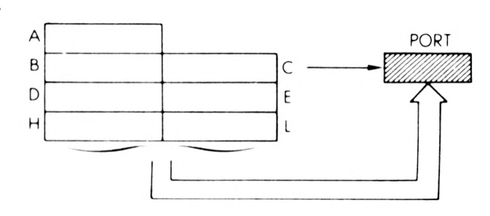
Description :

Le contenu du registre spécifié est écrit dans l'organe périphérique adressé par le contenu du registre C. r peut être n'importe lequel de :

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Le registre C fournit les bits A0 à A7 du bus adresse. Le registre B fournit les bits A8 à A15.

Chemin des données :



Durée : 3 cycles M ; 12 temps T ; 6 usec @ 2 MHz

Mode d'adressage : Externe.

Indicateurs :

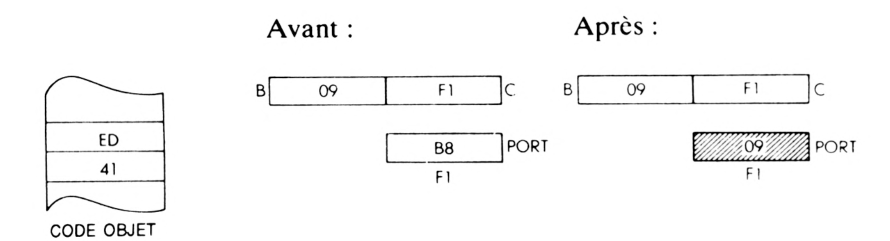
S	Z		H		P/V	N	C

(aucun effet)

Codes :

r:	A	B	C	D	E	H	L
	79	41	49	51	59	61	69

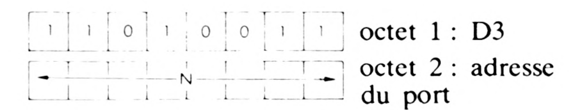
Exemple : OUT (C), B



OUT (N), A Sortie de l'accumulateur vers le port N.

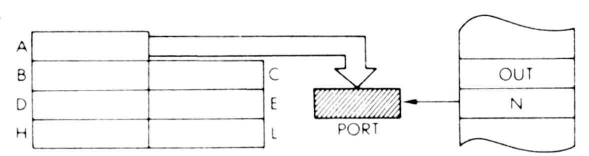
Fonction : (N) ← A

Format :



Description : Le contenu de l'accumulateur est écrit dans l'organe périphérique adressé par le contenu de l'emplacement mémoire suivant immédiatement le code opératoire.

Chemin des données :



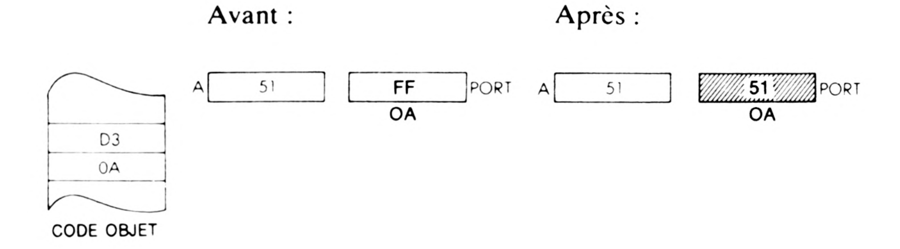
Durée : 3 cycles M ; 11 temps T ; 5,5 usec @ 2 MHz

Mode d'adressage : Externe.

Indicateurs :



Exemple : OUT (0A), A



OUTD

Sortie avec décrémentation.

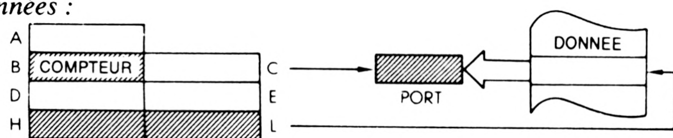
Fonction : (C) ← (HL) ; BC ← B − 1 ; HL ← HL − 1

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
1	0	1	0	1	0	1	1	octet 2 : AB

Description : Le contenu de l'emplacement mémoire adressé par le registre double HL est écrit dans l'organe périphérique adressé par le contenu du registre C. Le registre B et le registre double HL sont tous deux décrémentés. C fournit les bits A0 à A7 du bus adresse. B fournit (après décrémentation) les bits A8 à A15.

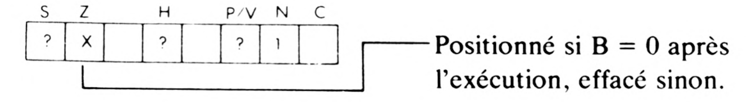
Chemin des données :



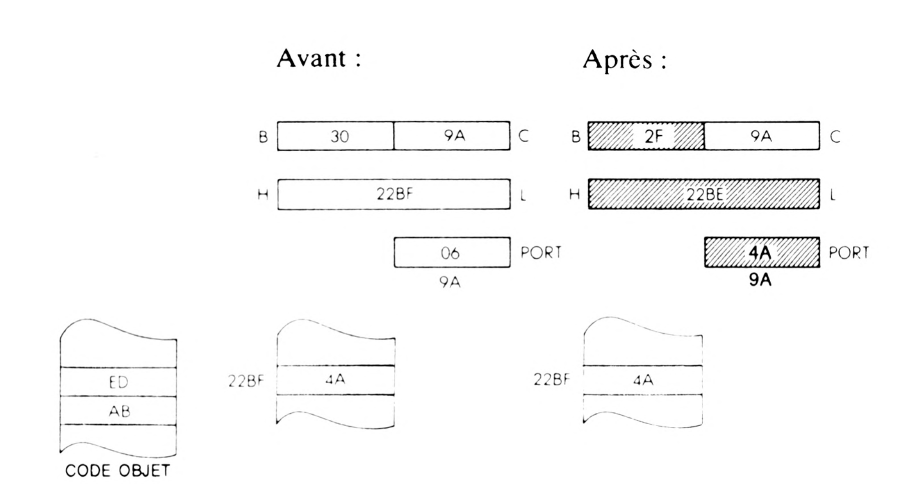
Durée : 4 cycles M ; 16 temps T ; 8 usec @ 2 MHz

Mode d'adressage : Externe.

Indicateurs :



Exemple : OUTD



OUTI

Sortie avec incrémentation.

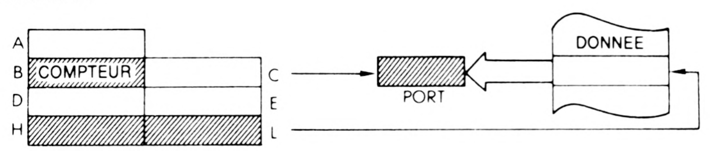
Fonction : (C) ← (HL) ; B ← B − 1 ; HL ← HL + 1

Format :

1	1	1	0	1	1	0	1	octet 1 : ED
1	0	1	0	0	0	1	1	octet 2 : A3

Description : Le contenu de l'emplacement mémoire adressé par le registre double HL est écrit dans l'organe périphérique adressé par le registre C. Le registre B est décrémenté et le registre double HL est incrémenté. C fournit les bits A0 à A7 du bus adresse. B (après décrémentation) fournit les bits A8 à A15.

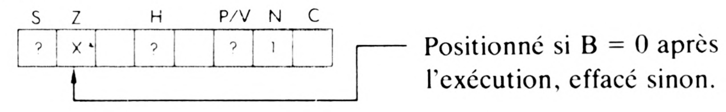
Chemin des données :



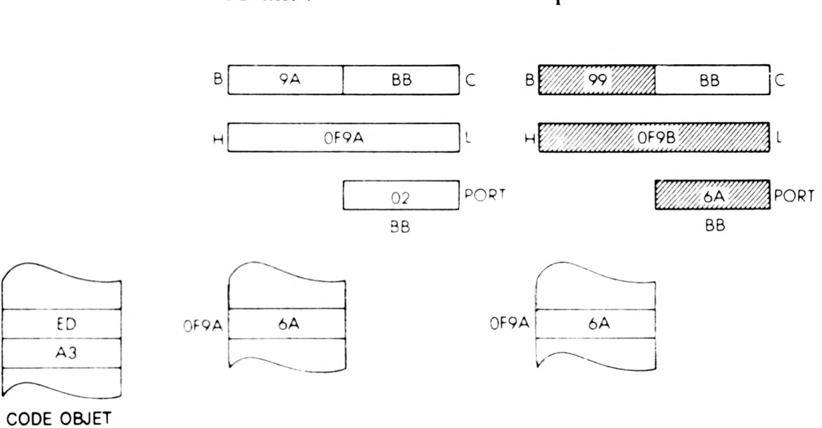
Durée : 4 cycles M ; 16 temps T ; 8 usec @ 2 MHz

Mode d'adressage : Externe.

Indicateurs :



Exemple : OUTI



POP qq Dépilement du registre double qq.

Fonction : $qq_{bas} \leftarrow (SP)$; $qq_{haut} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$

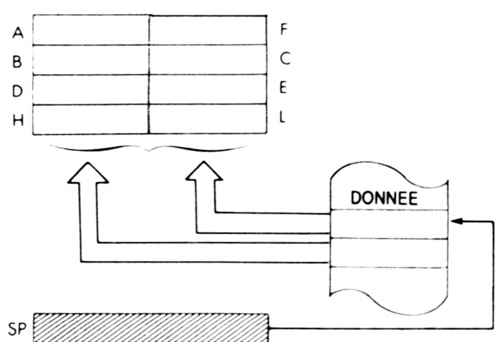
Format :

1	1	q	q	0	0	0	1

Description : Le contenu de l'emplacement mémoire adressé par le pointeur de pile est chargé dans la partie basse du registre double spécifié et ensuite le pointeur de pile est incrémenté. Le contenu de l'emplacement mémoire adressé par le pointeur de pile obtenu est chargé dans la partie haute du registre double, et le pointeur de pile est de nouveau incrémenté. qq peut être n'importe lequel de :

BC – 00 HL – 10
DE – 01 AF – 11


Chemin des données :



Durée : 3 cycles M ; 10 temps T ; 5 usec @ 2 MHz

Mode d'adressage : Indirect.

Codes :

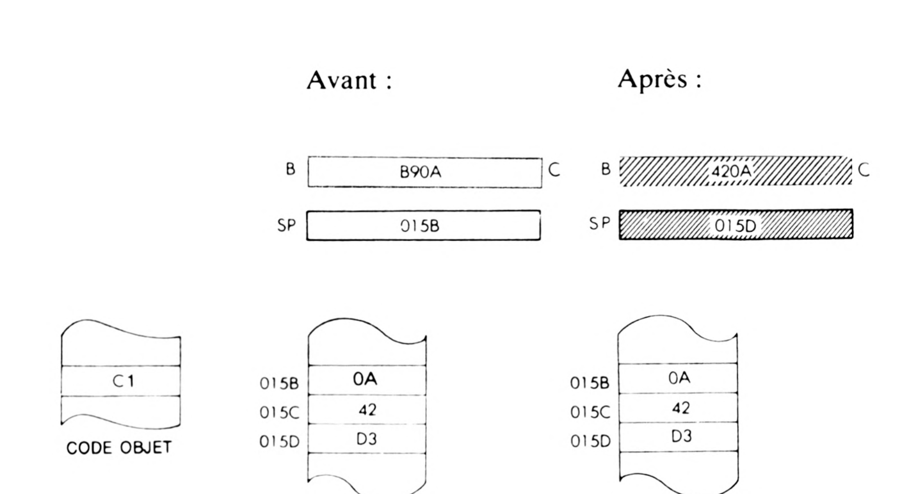
qq:	BC	DE	HL	AF
	C1	D1	E1	F1

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : POP BC



POP IX

Dépilement du registre IX.

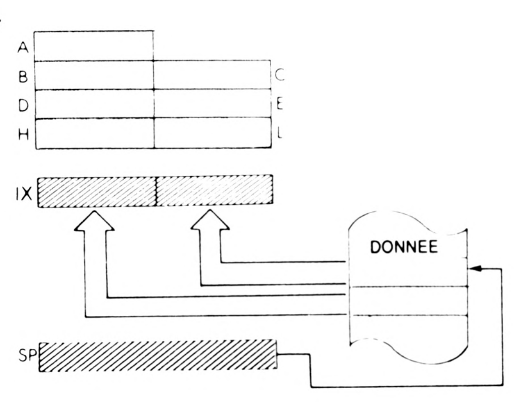
Fonction : $IX_{bas} \leftarrow (SP)$; $IX_{haut} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$

Format :

1	1	0	1	1	1	0	1	octet 1 : DD
1	1	1	0	0	0	0	1	octet 2 : E1

Description : Le contenu de l'emplacement mémoire adressé par le pointeur de pile est chargé dans la partie basse du registre IX et le pointeur de pile est incrémenté. Le contenu de l'emplacement mémoire adressé par le pointeur de pile obtenu est chargé dans le poids fort du registre IX et le pointeur de pile est de nouveau incrémenté.

Chemin des données :



Durée : 4 cycles M ; 14 temps T ; 7 usec @ 2 MHz

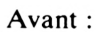
Mode d'adressage : Indirect.

Indicateurs :

S	Z		H		P/V	N	C

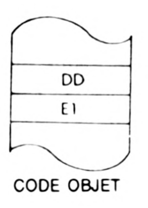
(aucun effet)

Exemple : POP IX

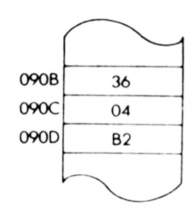


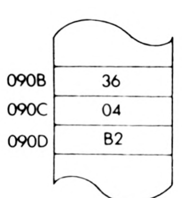
Après :

	Avant	Après
IX	0001	0436
SP	090B	090D



CODE OBJET





POP IY

Dépilement du registre IY.

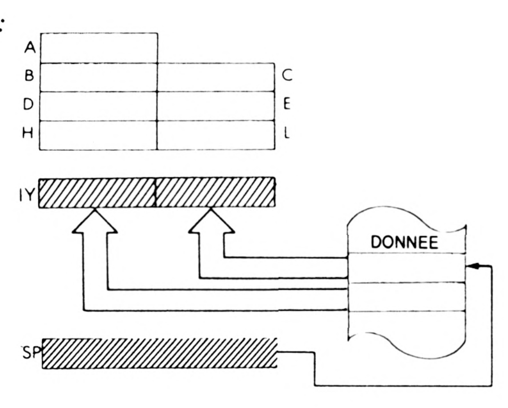
Fonction : $IY_{bas} \leftarrow (SP)$; $IY_{haut} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$

Format :

1	1	1	1	1	1	0	1	octet 1 : FD
1	1	1	0	0	0	0	1	octet 2 : E1

Description : Le contenu de l'emplacement mémoire adressé par le pointeur de pile est chargé dans le poids faible du registre IY et le pointeur de pile est incrémenté. Le contenu de l'emplacement mémoire adressé par le pointeur de pile obtenu est chargé dans le poids fort du registre IY, et le pointeur de pile est de nouveau incrémenté.

Chemin des données :



Durée : 4 cycles M ; 14 temps T ; 2 usec @ 2 MHz

Mode d'adressage : Indirect.

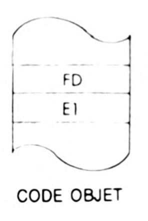
Indicateurs :

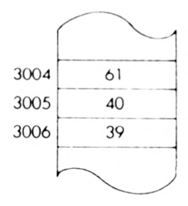
S	Z		H		P/V	N	C

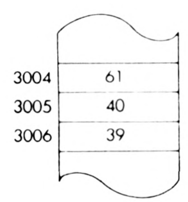
(aucun effet)

Exemple : POP IY

	Avant :	Après :
IY	032A	4061
SP	3004	3006







PUSH qq

Empilement du registre qq.

Fonction :

$(SP - 1) \leftarrow qq_{haut}$; $(SP - 2) \leftarrow qq_{bas}$
$SP \leftarrow SP - 2$

Format :

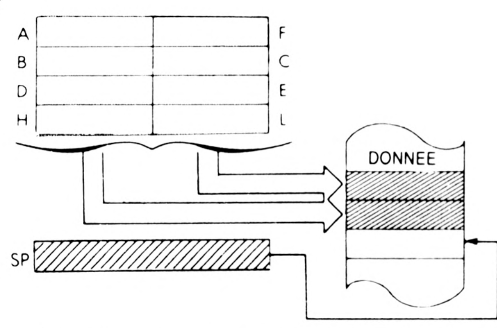
1	1	q	q	0	1	0	1

Description :

Le pointeur de pile est décrémenté et le contenu du poids fort du registre double spécifié est chargé dans l'emplacement mémoire adressé par le pointeur de pile. Le pointeur de pile est de nouveau décrémenté et le contenu du poids faible du registre double est chargé dans l'emplacement mémoire adressé alors par le pointeur de pile. qq peut être n'importe lequel de :

BC – 00   HL – 10
DE – 01   AF – 11

Chemin des données :



Durée :

3 cycles M ; 11 Temps T ; 6,5 usec @ 2 MHz

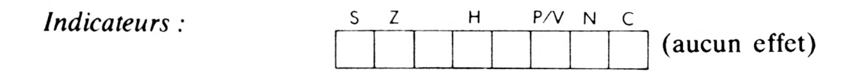
Mode d'adressage :

Indirect.

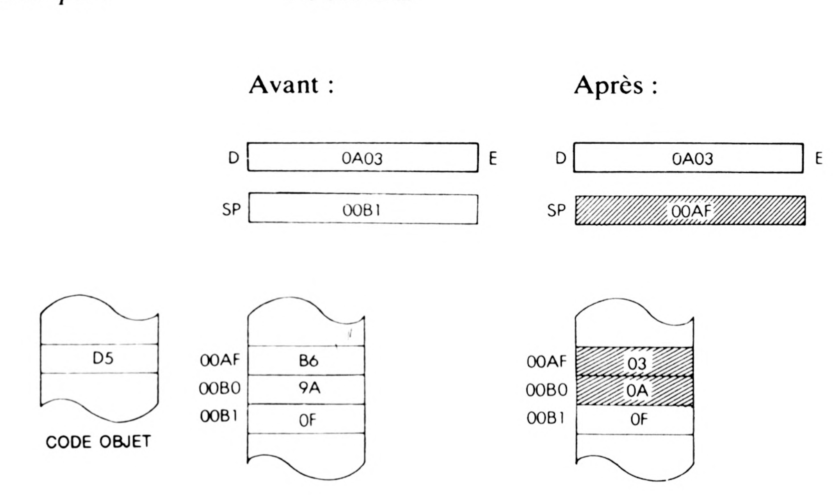
Codes :

qq:	BC	DE	HL	AF
	C5	D5	E5	F5

Indicateurs :



Exemple : PUSH DE



PUSH IX Empilement du registre IX.

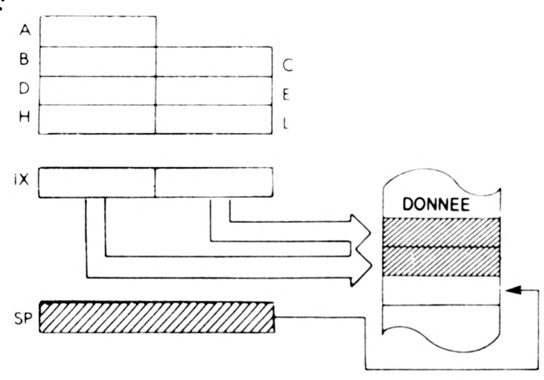
Fonction : $(SP - 1) \leftarrow IX_{haut}$; $(SP - 2) \leftarrow IX_{bas}$; $SP \leftarrow SP - 2$

Format :

1	1	0	1	1	1	0	1	octet 1 : DD
1	1	1	0	0	1	0	1	octet 2 : E5

Description : Le pointeur de pile est décrémenté et le contenu du poids fort du registre IX est chargé dans l'emplacement mémoire adressé par le pointeur de pile. Le pointeur de pile est de nouveau décrémenté et le contenu du poids faible du registre IX est chargé dans l'emplacement mémoire adressé par le pointeur de pile.

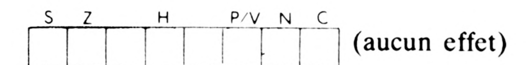
Chemin des données :



Durée : 4 cycles M ; 15 temps T ; 7,5 usec @ 2 MHz

Mode d'adressage : Indirect.

Indicateurs :



(aucun effet)

Exemple : PUSH IX

Avant :

IX 04A2

SP 0096

Après :

IX 04A2

SP 0094

CODE OBJET
DD
E5

Avant :

Adresse	Contenu
0094	8B
0095	9F
0096	04

Après :

Adresse	Contenu
0094	A2
0095	04
0096	04

PUSH IY

Empilement du registre IY.

Fonction :

$(SP - 1) \leftarrow IY_{haut}$; $(SP - 2) \leftarrow IY_{bas}$;
$SP \leftarrow SP - 2$

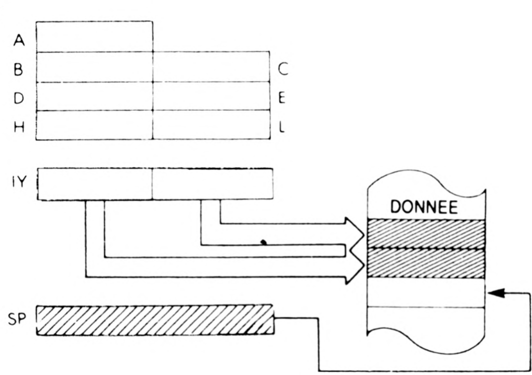
Format :

1	1	1	1	1	1	0	1	octet 1 : FD
1	1	1	0	0	1	0	1	octet 2 : E5

Description :

Le pointeur de pile est décrémenté et le contenu du poids fort du registre IY est chargé dans l'emplacement mémoire adressé par le pointeur de pile. Le pointeur de pile est de nouveau décrémenté et le contenu du poids faible du registre IY est chargé dans l'emplacement mémoire adressé par le pointeur de pile.

Chemin des données :



Durée : 3 cycles M ; 15 temps T ; 7,5 usec @ 2 MHz

Mode d'adressage : Indirect.

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : PUSH IY

Avant :

IY 90BF

SP 00B6

Après :

IY 90BF

SP 00B4

CODE OBJET
FD
E5

Avant :

Adresse	Contenu
00B4	FF
00B5	85
00B6	9D

Après :

Adresse	Contenu
00B4	BF
00B5	90
00B6	9D

RES b, s Effacement du bit b de l'opérande s.

Fonction : $s_b \leftarrow 0$

Format :

s:									
r	1	1	0	0	1	0	1	1	octet 1 : CB
	1	0	←	b	→	←	r	→	octet 2
(HL)	1	1	0	0	1	0	1	1	octet 1 : CB
	1	0	←	b	→	1	1	0	octet 2
(IX + d)	1	1	0	1	1	1	0	1	octet 1 : DD
	1	1	0	0	1	0	1	1	octet 2 : CB
	←			d				→	octet 3 : déplacement
	1	0	←	b	→	1	1	0	octet 4
(IY + d)	1	1	1	1	1	1	0	1	octet 1 : FD
	1	1	0	0	1	0	1	1	octet 2 : CB
	←			d				→	octet 3 : déplacement
	1	0	←	b	→	1	1	0	octet 4

b peut être n'importe lequel de :

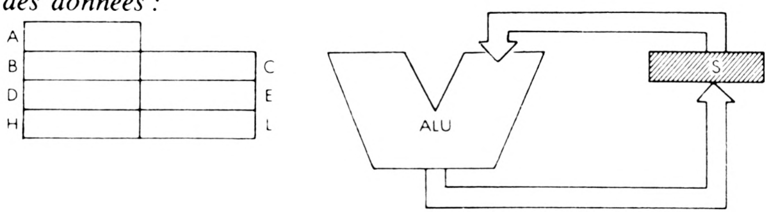
0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111

r peut être n'importe lequel de :

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Description : Le bit spécifié de l'emplacement déterminé par s est mis à zéro. s est défini dans la description des instructions BIT similaires.

Chemin des données :



Durée :

s:	*cycles M :*	*temps T :*	*usec @ 2 MHz:*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Mode d'adressage : r : implicite ; (HL) : indirect : (IX + d), (IY + d) : indexé.

Codes : RES b, r

CB

b: \ r:	A	B	C	D	E	H	L
0	87	80	81	82	83	84	85
1	8F	88	89	8A	8B	8C	8D
2	97	90	91	92	93	94	95
3	9F	98	99	9A	9B	9C	9D
4	A7	A0	A1	A2	A0	A4	A5
5	AF	A8	A9	AA	AB	AC	AD
6	B7	B0	B1	B2	B3	B4	B5
7	BF	B8	B9	BA	BB	BC	BD

RES b, (HL)

b:	0	1	2	3	4	5	6	7
CB	86	8E	96	9E	A6	AE	B6	BE

RES b, (IX + d) DD—
CB — d —
RES b, (IY + d) FD—

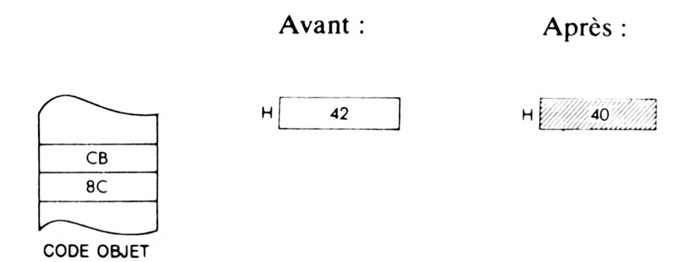
b:	0	1	2	3	4	5	6	7
	86	8E	96	9E	A6	AE	B6	BE

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : RES 1, H



RET

Retour de sous-programme.

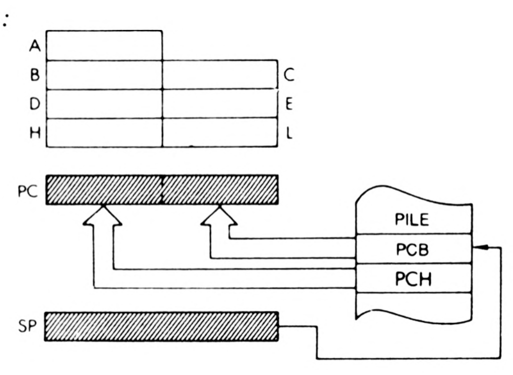
Fonction : $PC_{bas} \leftarrow (SP)$; $PC_{haut} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$

Format:

1	1	0	0	1	0	0	1	C9

Description : Le compteur ordinal est dépilé exactement comme décrit pour les instructions POP. L'instruction suivante est cherchée à l'emplacement pointé par PC.

Chemin des données :



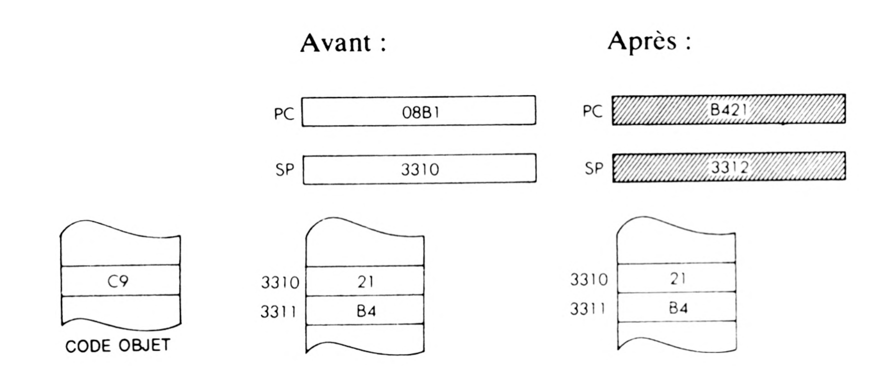
Durée : 3 cycles M ; 10 temps T ; 5 usec @ 2 MHz

Mode d'adressage : Indirect.

Indicateurs :

S	Z		H		P/V	N	C	
								(aucun effet)

Exemple : RET


Avant :
Après :
PC
08B1
SP
3310
PC
B421
SP
3312
C9
CODE OBJET
3310
21
3311
B4
3310
21
3311
B4

RET cc

Retour conditionnel de sous-programme.

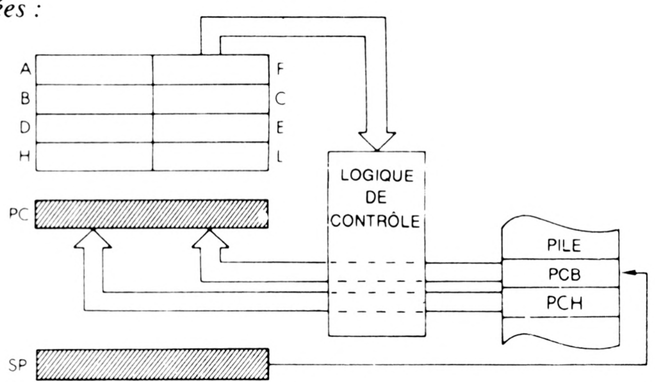
Fonction : Si cc vraie : $PC_{bas} \leftarrow (SP)$; $PC_{haut} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$

Format:

1	1	← cc →			0	0	0

Description : Si la condition est remplie, le compteur ordinal est dépilé comme décrit pour les instructions POP. L'instruction suivante est cherchée à l'adresse contenue dans PC. Si la condition n'est pas remplie, l'exécution des instructions continue en séquence.

Chemin des données :



cc peut être n'importe laquelle de :

NZ – 000	PO – 100
Z – 001	PE – 101
NC – 010	P – 110
C – 011	M – 111

Durée : Condition vraie : 3 cycles M ; 11 temps T ; 6,5 usec @ 2 MHz.
Condition fausse : 1 cycle M ; 5 temps T ; 2,5 usec @ 2 MHz

Mode d'adressage : Indirect.

Codes :

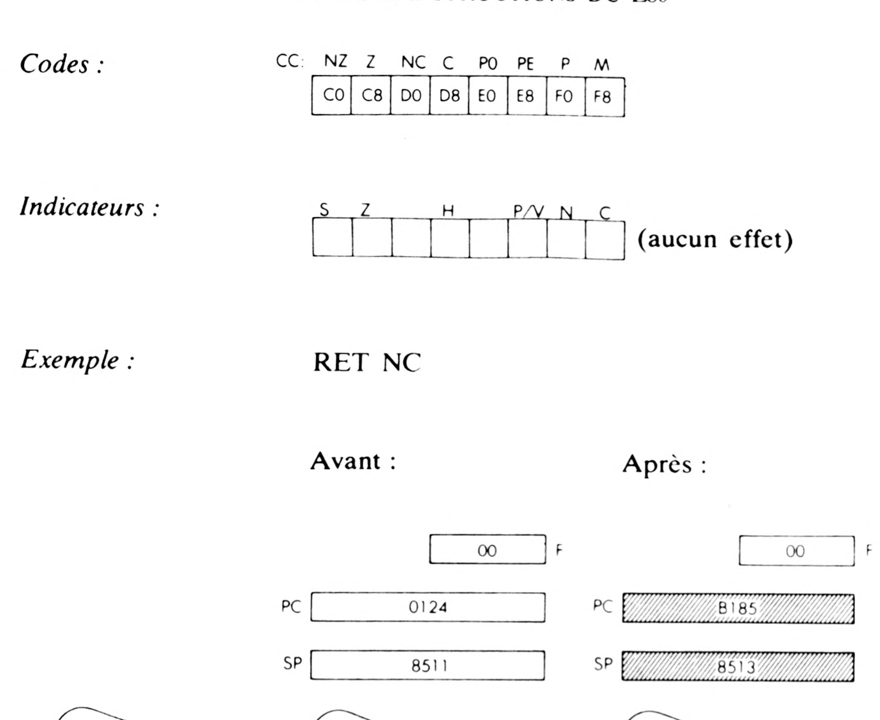
CC:	NZ	Z	NC	C	PO	PE	P	M
	C0	C8	D0	D8	E0	E8	F0	F8

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple : RET NC



RETI

Retour d'interruption.

Fonction :

$PC_{bas} \leftarrow (SP)$; $PC_{haut} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$

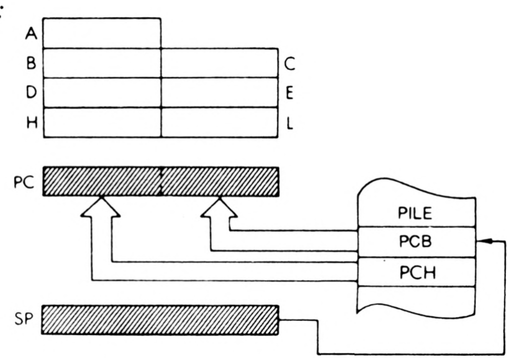
Format :

1	1	1	0	1	1	0	1	octet 1 : ED
0	1	0	0	1	1	0	1	octet 2 : 4D

Description :

Le compteur ordinal est dépilé comme décrit pour les instructions POP. Cette instruction est reconnue par les périphériques ZILOG comme la fin d'une routine d'interruption, pour permettre le contrôle correct de la hiérarchisation des interruptions. Une instruction EI doit être exécutée préalablement au RETI pour réautoriser les interruptions.

Chemin des données :



Durée :

4 cycles M ; 14 temps T ; 7 usec @ 2 MHz

Mode d'adressage :

Indirect.

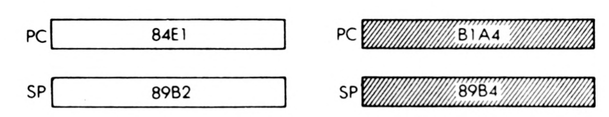
Indicateurs :

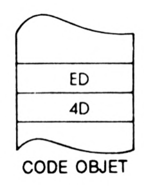
S	Z		H		P/V	N	C

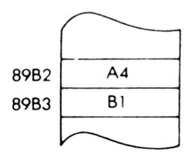
(aucun effet)

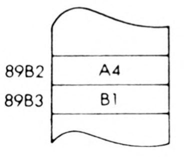
Exemple : RETI

Avant : Après :









RETN

Retour d'interruption non masquable.

Fonction :

$PC_{bas} \leftarrow (SP)$; $PC_{haut} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$; $IFF1 \leftarrow IFF2$

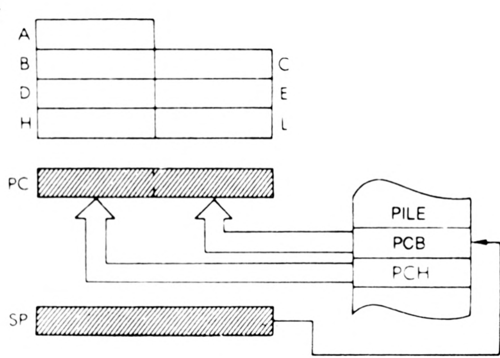
Format :

1	1	1	0	1	1	0	1	octet 1 : ED
0	1	0	0	0	1	0	1	octet 2 : 45

Description :

Le compteur ordinal est dépilé comme décrit pour les instructions POP. Ensuite le contenu de IFF2 (bascule de sauvegarde) est recopié dans IFF1 pour restaurer l'état de la bascule d'autorisation des interruptions avant l'interruption non masquable.

Chemin des données :

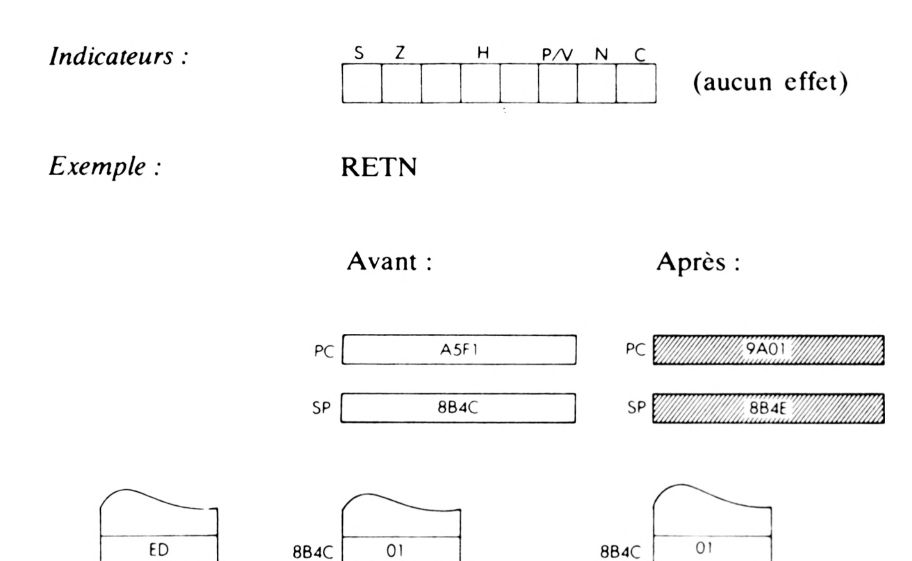


Durée :

4 cycles M ; 14 temps T ; 7 usec @ 2 MHz

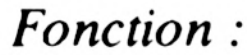
Mode d'adressage :

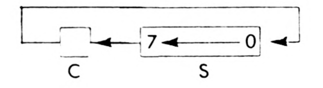
Indirect.



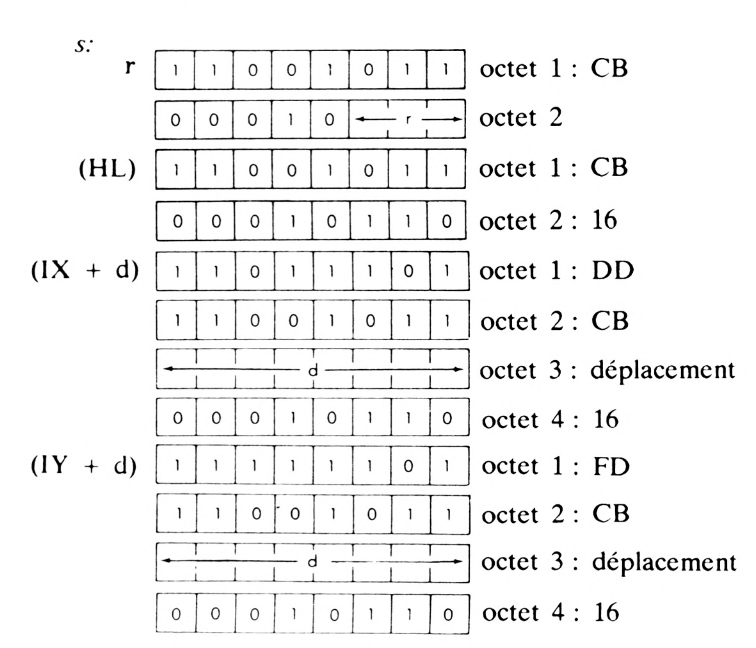
Indicateurs :
S Z H P/V N C
(aucun effet)
Exemple :
RETN
Avant :
Après :
PC A5F1
SP 8B4C
PC 9A01
SP 8B4E
ED
45
CODE OBJET
8B4C 01
8B4D 9A
8B4C 01
8B4D 9A

RL s Rotation de l'opérande s à gauche à travers le report.





Format :

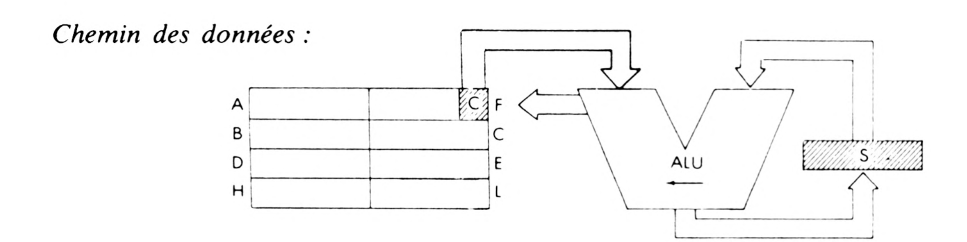


r peut être n'importe lequel de :

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Description : Le contenu de l'emplacement spécifié par l'opérande est décalé vers la gauche d'un bit. Le contenu de l'indicateur de report va dans le bit 0 et le contenu du bit 7 va dans l'indicateur de report. Le résultat final est rangé à nouveau dans l'emplacement d'origine. s est défini dans la description des instructions RLC similaires.

Chemin des données :

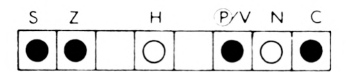


Durée :

s:	*cycles M :*	*temps T :*	*usec @ 2 MHz:*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Mode d'adressage : r : implicite ; (HL) : indirect ; (IX + d), (IY + d) : indexé.

Codes : RL r

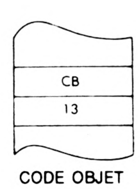


Indicateurs :

S	Z		H		P/V	N	C
●	●		○		●	○	●

C est positionné par le bit 7 de l'opérande source.

Exemple : RL E



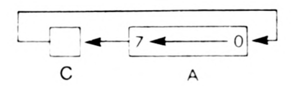
Avant : Après :



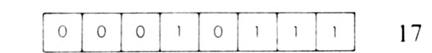
RLA

Rotation de l'accumulateur à gauche à travers le report.

Fonction :



Format :

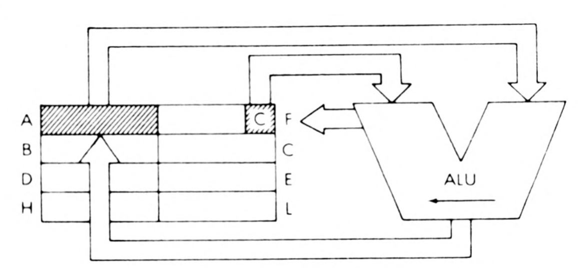


17

Description :

Le contenu de l'accumulateur est décalé vers la gauche d'un bit. Le contenu de l'indicateur de report va dans le bit 0 et le contenu initial du bit 7 va dans l'indicateur de report (rotation sur 9 bits).

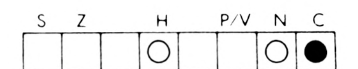
Chemin des données :



Durée : 1 cycle M ; 4 temps T ; 2 usec @ 2 MHz

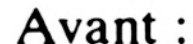
Mode d'adressage : Implicite.

Indicateurs :



C est positionné par le bit 7 de A

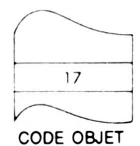
Exemple : RLA


Avant :


Après :

A | 0F | 01 | F

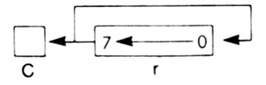




RLC r

Rotation du registre r à gauche sans le report.

Fonction :



Format :

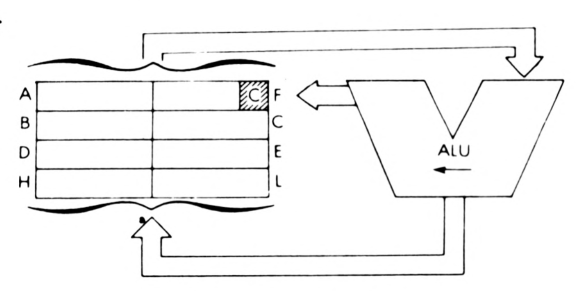
1	1	0	0	1	0	1	1	octet 1 : CB
0	0	0	0	0	←	r	→	octet 2

Description :

Le contenu du registre spécifié est décalé circulairement vers la gauche. Le contenu initial du bit 7 va dans l'indicateur de report en même temps que dans le bit 0. r peut être n'importe lequel de :

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Chemin des données :



Durée : 2 cycles M ; 8 temps T ; 4 usec @ 2 MHz

Mode d'adressage : Implicite.

Codes : r:

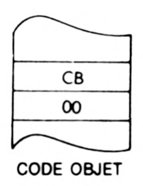
	A	B	C	D	E	H	L
CB-	07	00	01	02	03	04	05

Indicateurs :

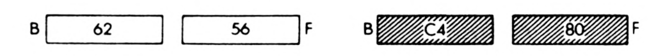
S	Z		H		Ⓟ/V	N	C
●	●		○		●	○	●

C est positionné par le bit 7 du registre source

Exemple : RLC B



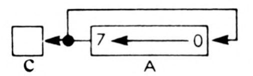
Avant : Après :



RLCA

Rotation de l'accumulateur à gauche sans le report.

Fonction :



Format :

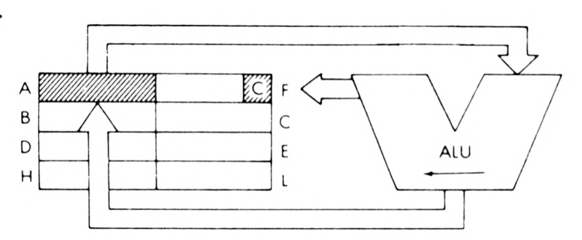
0	0	0	0	0	1	1	1

07

Description :

Le contenu de l'accumulateur est décalé circulairement vers la gauche d'un bit. Le contenu initial du bit 7 va dans l'indicateur de report en même temps que dans le bit 0.

Chemin des données :



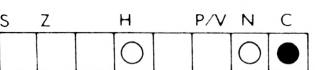
Durée :

1 cycle M ; 4 temps T ; 2 usec @ 2 MHz

Mode d'adressage :

Implicite.

Indicateurs :

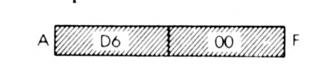


C prend la valeur de l'ancien bit 7 de A

Exemple :

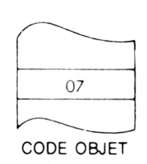
RLCA

Avant :



Après :

A | D6 | 00 | F

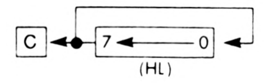


Note : cette instruction est identique à RLC A sauf en ce qui concerne les indicateurs. Elle existe pour la comptabilité avec le 8080.

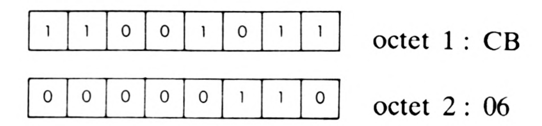
RLC (HL)

Rotation à gauche sans le report de l'emplacement mémoire (HL).

Fonction :



Format :



octet 1 : CB

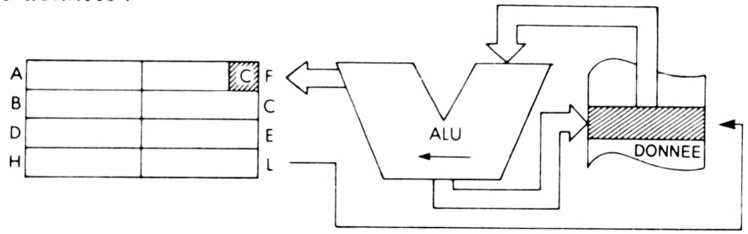
octet 2 : 06

Description :

Le contenu de l'emplacement mémoire adressé par le contenu du registre double HL est décalé circulairement vers la gauche d'un bit et le résultat est rangé à nouveau à cet emplacement. Le contenu du bit 7 va dans l'indicateur de report en même temps que dans le bit 0.



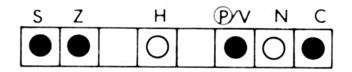
Chemin des données :



Durée : 4 cycles M ; 15 temps T ; 7,5 usec @ 2 MHz

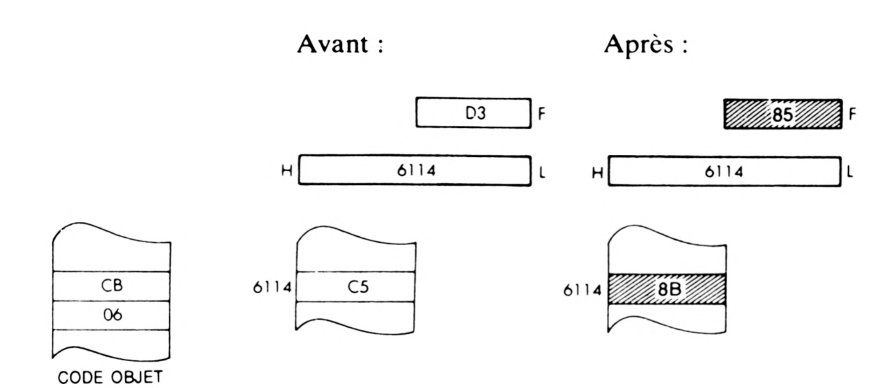
Mode d'adressage : Indirect.

Indicateurs :



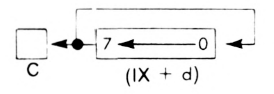
C est positionné par le bit 7 de l'emplacement mémoire

Exemple : RLC (HL)

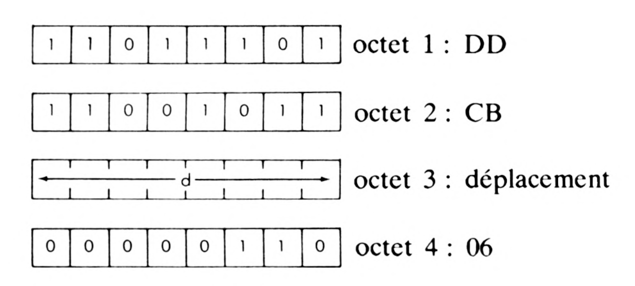


RLC (IX + d) Rotation à gauche sans le report de l'emplacement mémoire (IX + d).

Fonction :

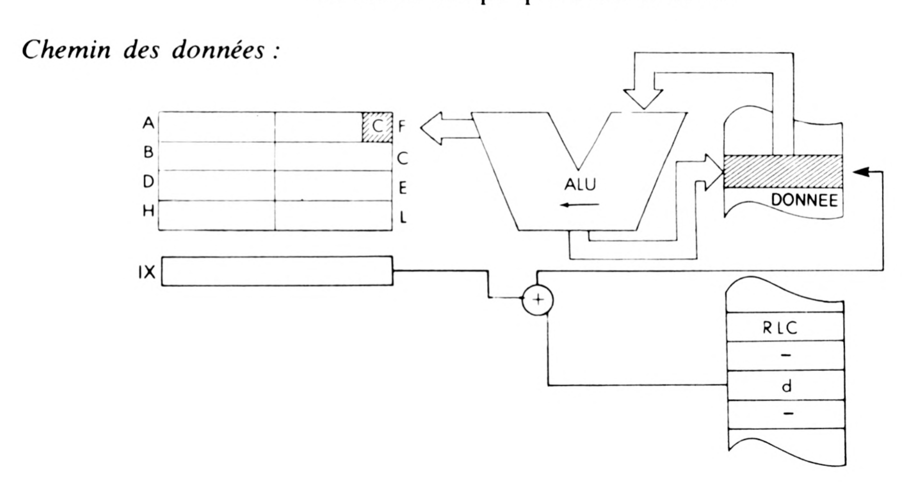


Format :



Description : Le contenu de l'emplacement mémoire adressé par le contenu du registre IX plus le déplacement fourni est décalé circulairement vers la gauche et le résultat est rangé à nouveau dans cet emplacement. Le contenu du bit 7 va dans l'indicateur de report en même temps que dans le bit 0.

Chemin des données :



Durée : 6 cycles M ; 23 temps T ; 11,5 usec @ 2 MHz

Mode d'adressage : Indexé.

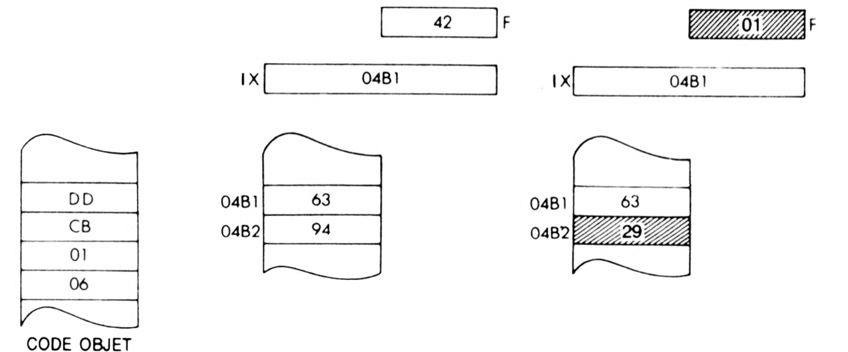
Indicateurs :



C est positionné par le bit 7 de l'emplacement mémoire

Exemple : RLC (IX + 1)

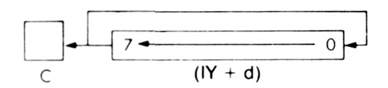
Avant : Après :



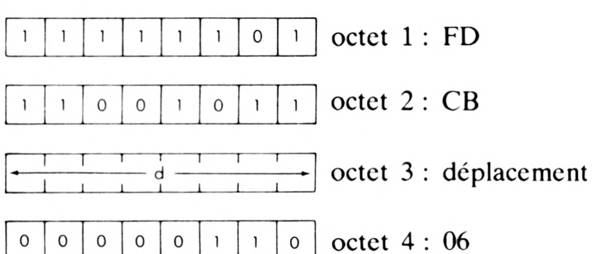
RLC (IY + d) Rotation à gauche sans le report de l'emplacement mémoire (IY + d).



Fonction :

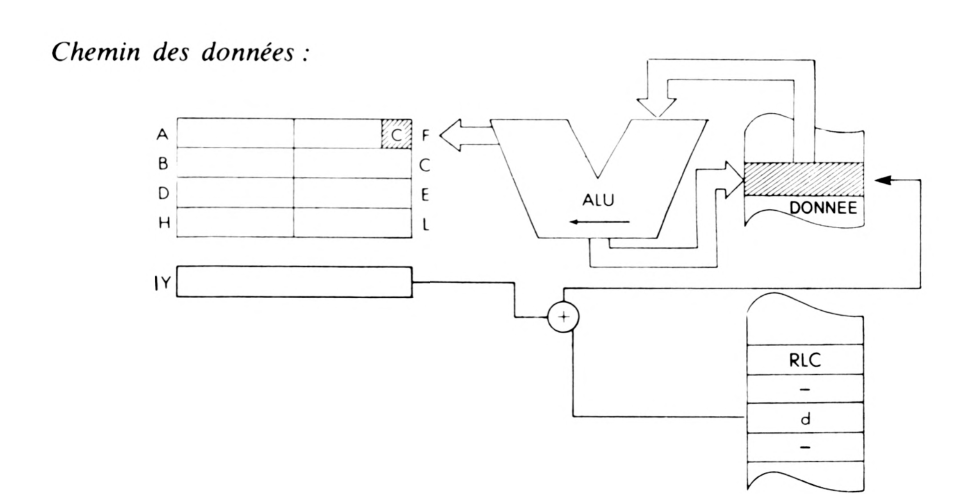


Format :



Description : Le contenu de l'emplacement mémoire adressé par le contenu du registre IY plus le déplacement fourni est décalé circulairement vers la gauche et le résultat est rangé à nouveau dans cet emplacement. Le contenu du bit 7 va dans l'indicateur de report en même temps que dans le bit 0.

Chemin des données :



Durée : 6 cycles M ; 23 temps T ; 11,5 usec @ 2 MHz

Mode d'adressage : Indexé.

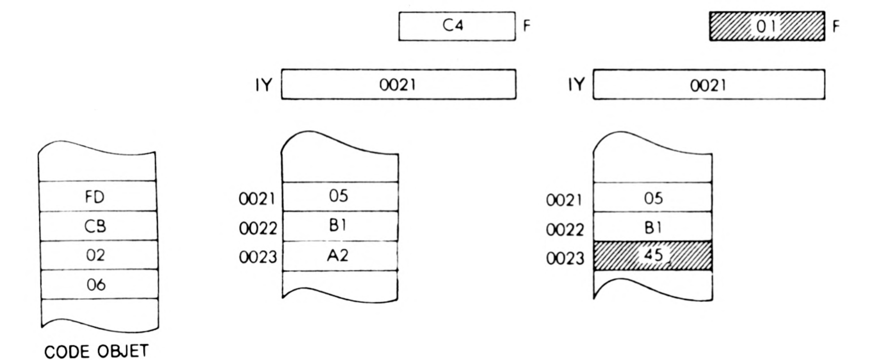
Indicateurs :

S	Z		H		P/V	N	C
●	●		○		●	○	●

C est positionné par le bit 7 de l'emplacement mémoire

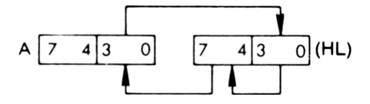
Exemple : RLC (IY + 2)

Avant : Après :



RLD Rotation décimale à gauche.

Fonction :

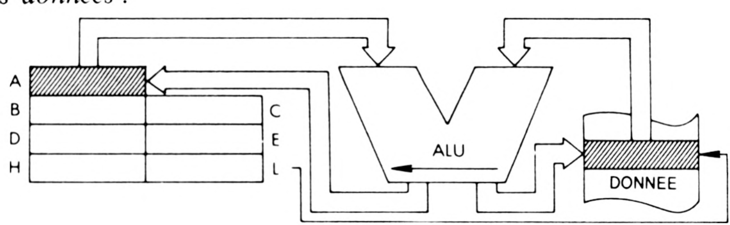


Format :

1	1	1	0	1	1	0	1	octet 1 : ED
0	1	1	0	1	1	1	1	octet 2 : 6F

Description : Les 4 bits de poids faible de l'emplacement mémoire adressé par le contenu de HL sont placés dans les bits de poids fort de ce même emplacement. Les 4 bits de poids fort sont placés dans les 4 bits de poids faible de l'accumulateur. Le poids faible de l'accumulateur est placé dans les 4 bits de poids faible de l'emplacement mémoire initial. Toutes ces opérations se produisent simultanément.

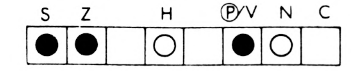
Chemin des données :



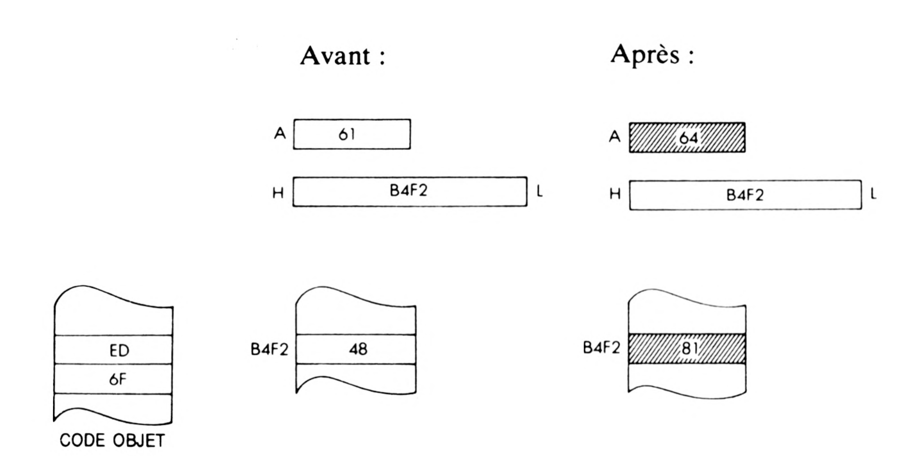
Durée : 5 cycles M ; 18 temps T ; 9 usec @ 2 MHz

Mode d'adressage : Indirect.

Indicateurs :



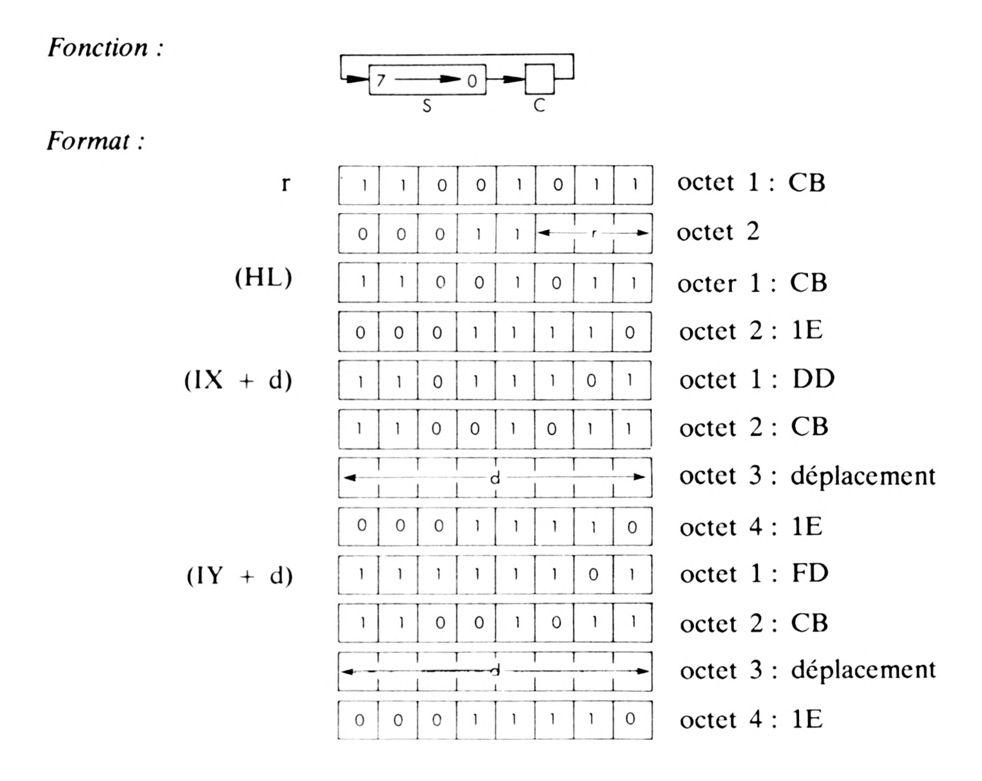
Exemples : **RLD**



RR s Rotation à droite à travers le report.

Fonction :

Format :

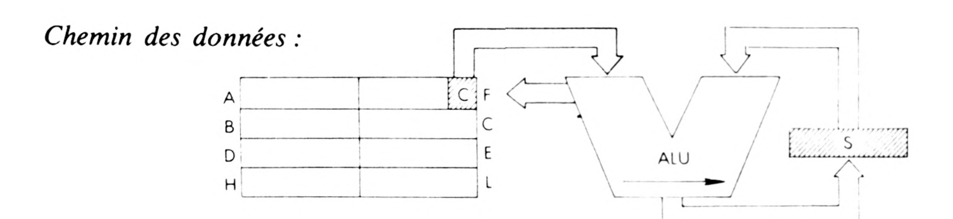


r peut être n'importe lequel de :

A – 111
B – 000
C – 001
D – 010
E – 011
H – 100
L – 101

Description : Le contenu de l'emplacement déterminé par l'opérande spécifié est décalé vers la droite. Le contenu de l'indicateur de report va dans le bit 7 et le contenu du bit 0 va dans l'indicateur de report. Le résultat final est rangé à nouveau dans l'emplacement initial. s est défini dans la description des instructions RLC similaires.

Chemin des données :

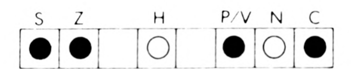


Durée :

s:	*cycles M :*	*temps T :*	*usec @ 2 MHz:*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Mode d'adressage : r : implicite ; (HL) : indirect : (IX + d), (IY + d) : indexé.

Codes : RR r :



r:	A	B	C	D	E	H	L
CB-	1F	18	19	1A	1B	1C	1D

Indicateurs :

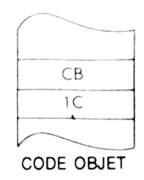
S	Z		H		P/V	N	C
●	●		○		●	○	●

C est positionné par le bit 0 de la donnée source

Exemple : RR H

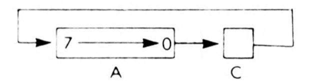
Avant : H 6B 41 F

Après : H B5 81 F

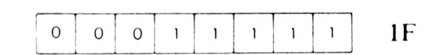


RRA Rotation de l'accumulateur à droite à travers le report.

Fonction :



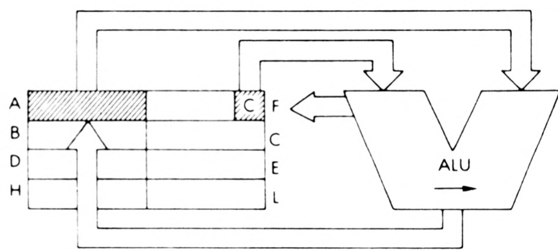
Format :



1F

Description : Le contenu de l'accumulateur est décalé vers la droite d'un bit. Le contenu de l'indicateur de report va dans le bit 7 et le contenu du bit 0 va dans l'indicateur de report (rotation sur 9 bits).

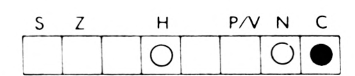
Chemin des données :



Durée : 1 cycle M ; 4 temps T ; 2 usec @ MHz

Mode d'adressage : Implicite.

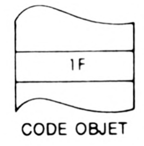
Indicateurs :

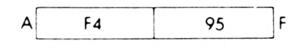


C est positionné par le bit 0 de A

Exemple : RRA

Avant : Après :





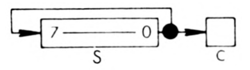


Note : cette instruction est identique à RL A sauf en ce qui concerne les indicateurs. Elle existe pour la comptabilité avec le 8080.

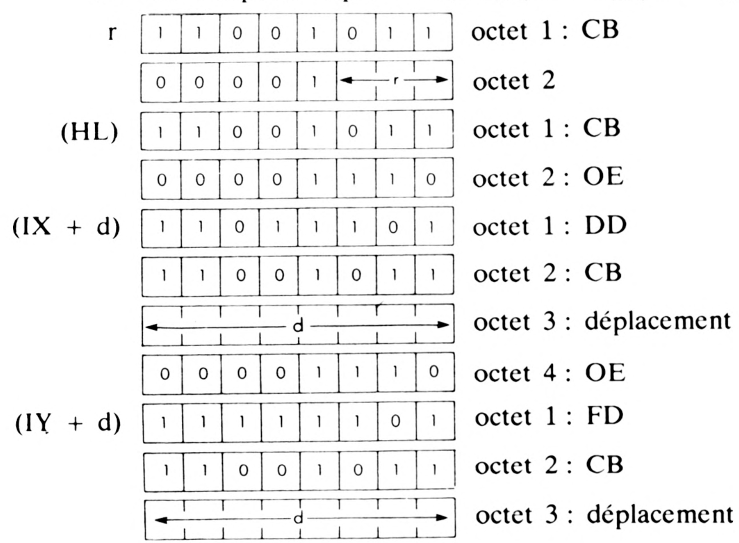
RRC s

Rotation à droite sans le report.

Fonction :



Format : s : est n'importe lequel de : (HL), (IX + d), (IY + d).

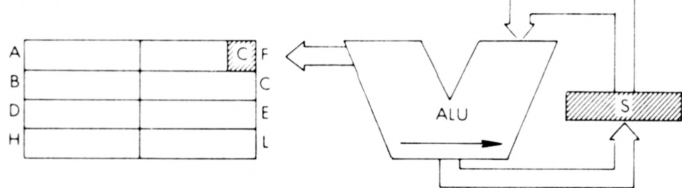


r peut être n'importe lequel de :

A – 111	E – 011
B – 000	H – 100
C – 011	L – 101
D – 010	

Description : Le contenu de l'emplacement déterminé par l'opérande spécifié est décalé circulairement vers la droite et le résultat est rangé à nouveau dans l'emplacement initial. Le contenu du bit 0 va dans l'indicateur de report en même temps que dans le bit 7. s est défini dans la description des instructions RLC similaires.

Chemin des données :



Durée :

s:	cycles M :	temps T :	μsec @ 2 MHz:
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Mode d'adressage : r : implicite ; (HL) : indirect ; (IX + d), (IY + d) : indexé.

Codes : RRC r

r:	A	B	C	D	E	H	L
CB-	0F	08	09	0A	0B	0C	0D

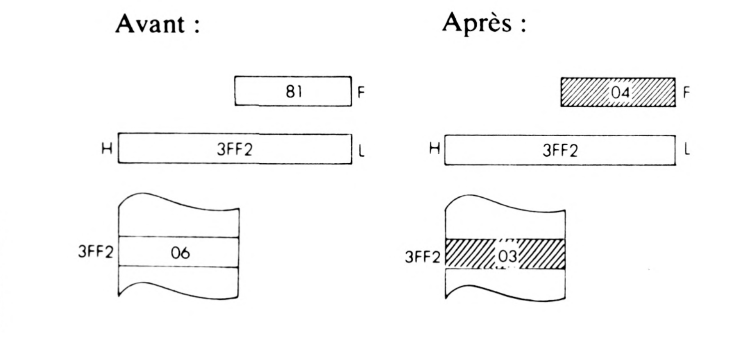
Indicateurs :

S	Z		H		(P)/V	N	C
●	●		○		●	○	●

C est positionné par le bit 0 de la donnée source

Exemple : RRC (HL)

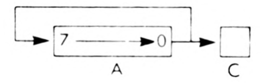
Avant : Après :



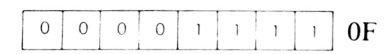
RRCA

Rotation de l'accumulateur à droite sans le report.

Fonction :



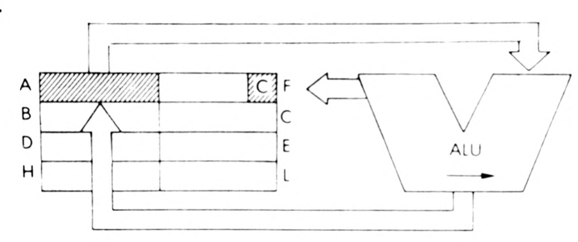
Format :



0F

Description : Le contenu de l'accumulateur est décalé circulairement vers la droite d'un bit. Le contenu du bit 0 va dans l'indicateur de report en même temps que dans le bit 7.

Chemin des données :



Durée : 1 cycles M ; 4 temps T ; 2 usec @ 2 MHz

Mode d'adressage : Implicite.

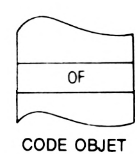
Indicateurs :

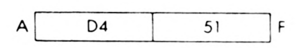
S	Z		H		P/V	N	C
			○			○	●

C est positionné par le bit 0 de A

Exemple : RRCA

Avant : Après :







RRD

Rotation décimale à droite.

Fonction :

A | 7 4 | 3 0 | → | 7 4 | 3 0 | (HL)

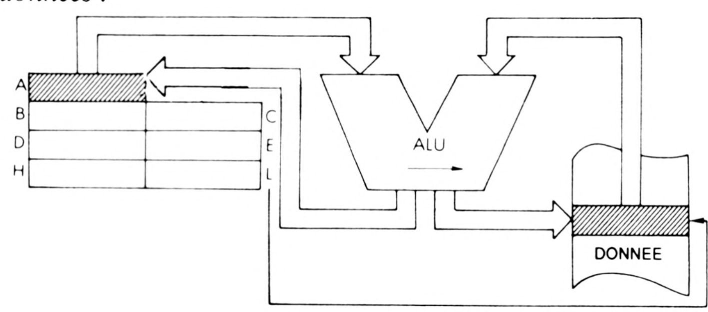
Format :

1	1	1	0	1	1	0	1	octet 1 : ED
0	1	1	0	0	1	1	1	octet 2 : 67

Description :

Les 4 bits de poids fort de l'emplacement mémoire adressé par le contenu du registre double HL sont placés dans les 4 bits de poids faible de cet emplacement. Les 4 bits de poids faible sont placés dans les 4 bits de poids faible de l'accumulateur. Les bits de poids faible de l'accumulateur sont placés dans les 4 bits de poids fort de l'emplacement mémoire initial. Toutes ces opérations se produisent simultanément.

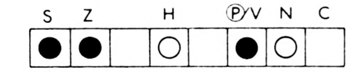
Chemin des données :



Durée : 5 cycles M ; 18 temps T ; 9 usec @ 2 MHz

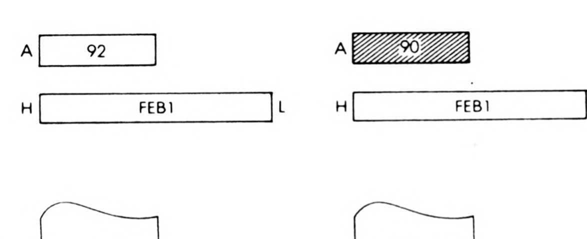
Mode d'adressage : Indirect.

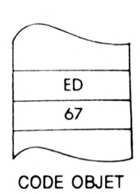

Indicateurs :

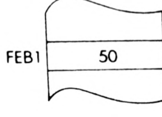


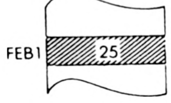
Exemple : RRD

Avant : Après :









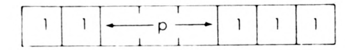
RST p

Redépart en p.

Fonction :

$(SP - 1) \leftarrow PC_{haut}$; $(SP - 2) \leftarrow PC_{bas}$;
$SP \leftarrow SP - 2$; $PC_{haut} \leftarrow 0$; $PC_{bas} \leftarrow p$

Format:



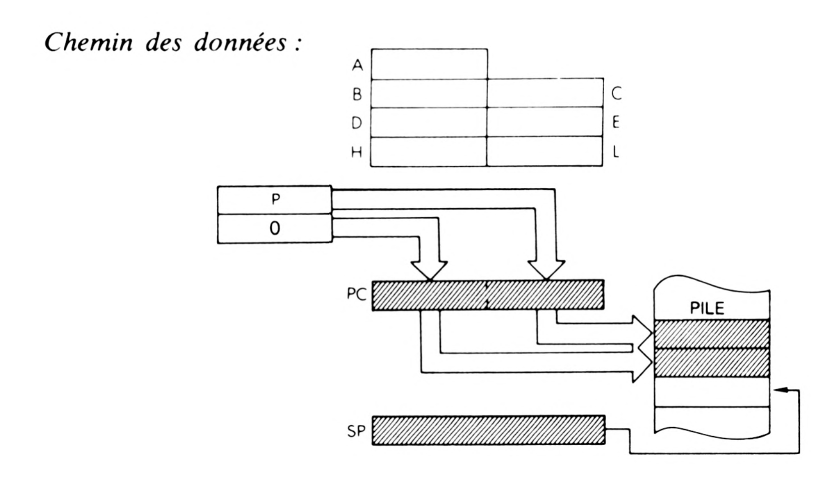
Description :

Le contenu du compteur ordinal est empilé comme décrit pour les instructions PUSH. La valeur spécifiée pour p est ensuite chargée dans le compteur ordinal et l'instruction suivante est cherchée à cette nouvelle adresse. p peut être n'importe lequel de :

00H – 000	20H – 100
08H – 001	28H – 101
10H – 010	30H – 110
18H – 011	38H – 111

Cette instruction effectue un saut à l'une parmi huit adresses en mémoire basse et ne nécessite qu'un seul octet. Elle peut être utilisée pour réagir rapidement à une interruption.

Chemin des données :



Durée : 3 cycles M ; 11 temps T ; 5,5 usec @ 2 MHz

Mode d'adressage : Indirect.

Codes : p:

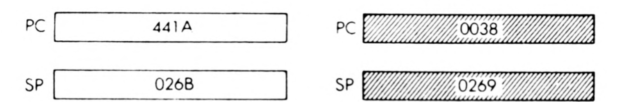


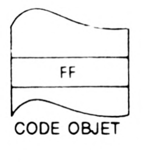
Indicateurs :

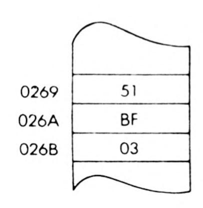
S Z H P/V N C

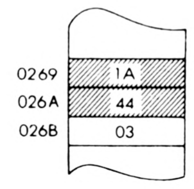
(aucun effet)

Exemple : RST 38H









SBC A, s Soustraire de l'accumulateur l'opérande spécifié et le report.

Fonction : A ← A − s − C

Format : s : peut être r, n, (HL), (IX + d), ou (IY + d)

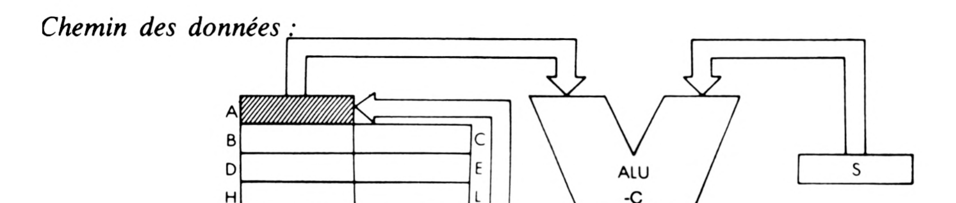
r	1 0 0 1 1 ←r→	
n	1 1 0 1 1 1 1 0	octet 1 : DE
	←n→	octet 2 : donnée immédiate
(HL)	1 0 0 1 1 1 1 0	octet 1 : 9E
(IX + d)	1 1 0 1 1 1 0 1	octet 1 : DD
	1 0 0 1 1 1 1 0	octet 2 : 9E
	←d→	octet 3 : déplacement
(IY + d)	1 1 1 1 1 1 0 1	octet 1 : FD
	1 0 0 1 1 1 1 0	octet 2 : 9E
	←d→	octet 3 : déplacement

r peut être n'importe lequel de :

A − 111	E − 011
B − 000	H − 100
C − 001	L − 101
D − 010	

Description : L'opérande spécifié s, augmenté du contenu de l'indicateur de report, est soustrait du contenu de l'accumulateur et le résultat est placé dans l'accumulateur. s est défini dans la description des instructions similaires ADD.

Chemin des données :

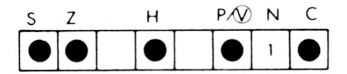


Durée :

s:	*cycles M :*	*temps T :*	*usec @ 2 MHz:*
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Mode d'adressage : r : implicite ; n : immédiat ; (HL) : indirect ; (IX + d), (IY + d) : indexé.

Codes : SBC A, r

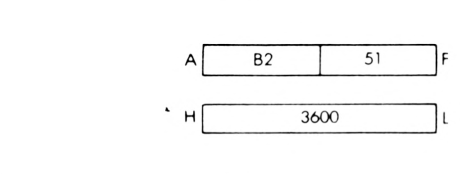


Indicateurs :

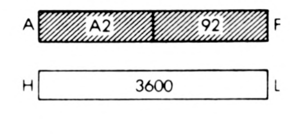
S	Z		H		P/V	N	C
●	●		●		●	1	●

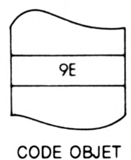
Exemple : SBC A, (HL)

Avant :

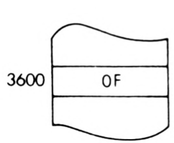


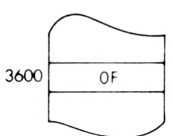
Après :





CODE OBJET





SBC HL, ss

Soustraire de HL le registre double ss et le report.

Fonction : HL ← HL − ss − C

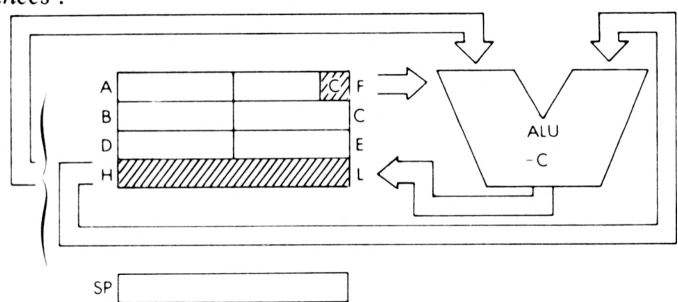
Format :

1	1	1	0	1	1	0	1	octet 1 : ED
0	1	s	s	0	0	1	0	octet 2

Description : Le contenu du registre double spécifié augmenté du contenu de l'indicateur de report est soustrait du contenu du registre double HL et le résultat est rangé à nouveau dans HL. ss peut être n'importe lequel de :

BC − 00 HL − 10
DE − 01 SP − 11

Chemin des données :



Durée : 4 cycles M ; 15 temps T ; 7,5 usec @ 2 MHz

Mode d'adressage : Implicite.

Codes :

ss:	BC	DE	HL	SP
ED	42	52	62	72



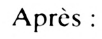
Indicateurs :

S	Z		H		P/V	N	C
●	●		●		●	1	●

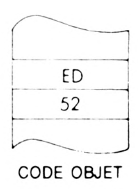
H est positionné s'il y a report du bit 12
C est positionné s'il y a report

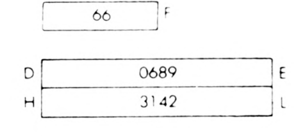
Exemple : SBC HL, DE

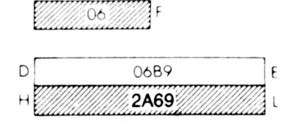
Avant :



Après :







SCF

Positionnement de l'indicateur de report..

Fonction : C ← 1

Format :

0	0	1	1	0	1	1	1

37

Description : L'indicateur de report est positionné.

Durée : 1 cycle M ; 4 temps T ; 2 usec @ 2 MHz

Mode d'adressage : Implicite

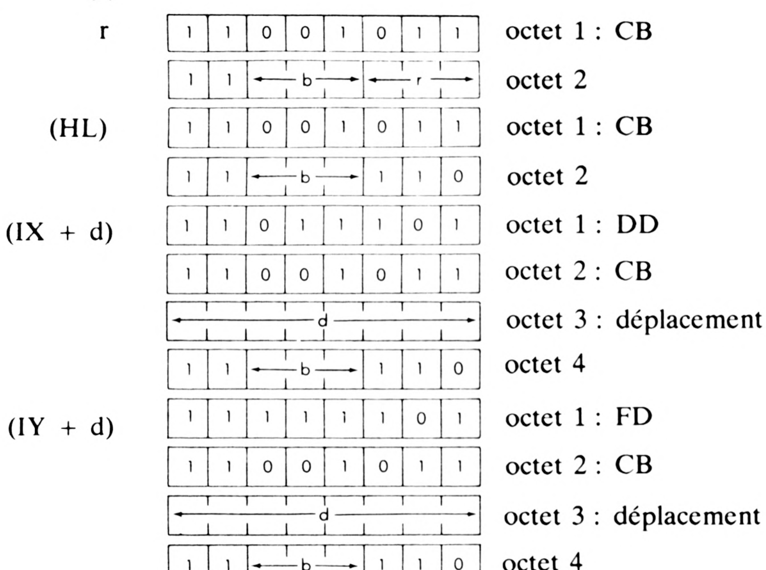
Indicateurs :

S	Z		H		P/V	N	C
			O			O	1

SET b, s Positionnement du bit b de l'opérande s.

Fonction : $s_b \leftarrow 1$

Format : s :



r peut être n'importe lequel de :

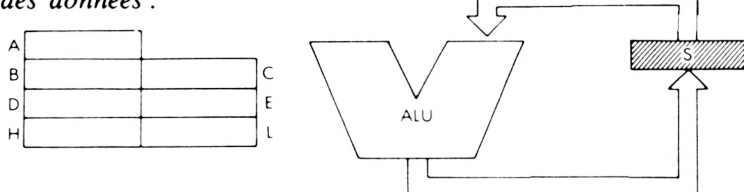
A – 111	E – 001
B – 000	H – 100
C – 001	L – 101
D – 010	

b peut être n'importe lequel de :

0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111

Description : Le bit spécifié de l'emplacement déterminé par s est positionné. s est défini dans la description des instructions similaires BIT.





Durée :

s:	cycles M :	temps T :	usec @ 2 MHz:
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Mode d'adressage : r : implicite ; (HL) : indirect ; (IX + d), (IY + d) : indexé.

Codes : SET b, r

CB–

b: r:	A	B	C	D	E	H	L
0	C7	C0	C1	C2	C3	C4	C5
1	CF	**C8**	C9	CA	CB	CC	CD
2	D7	D0	D1	D2	D3	D4	D5
3	DF	D8	D9	DA	DB	DC	DD
4	E7	E0	E1	E2	E3	E4	E5
5	EF	E8	E9	EA	EB	EC	ED
6	F7	F0	F1	F2	F3	F4	F5
7	FF	F3	F9	FA	FB	FC	FD

SET b, (HL)

SET b, (IX + d)

SET b, (IY + d)

b:	0	1	2	3	4	5	6	7
	C6	CE	Db	**D6**	**DE**	EE	F6	FE

Indicateurs :

S	Z		H		P/V	N	C

(aucun effet)

Exemple SET 7, A

Avant :

A: 61

Après :

A: E1

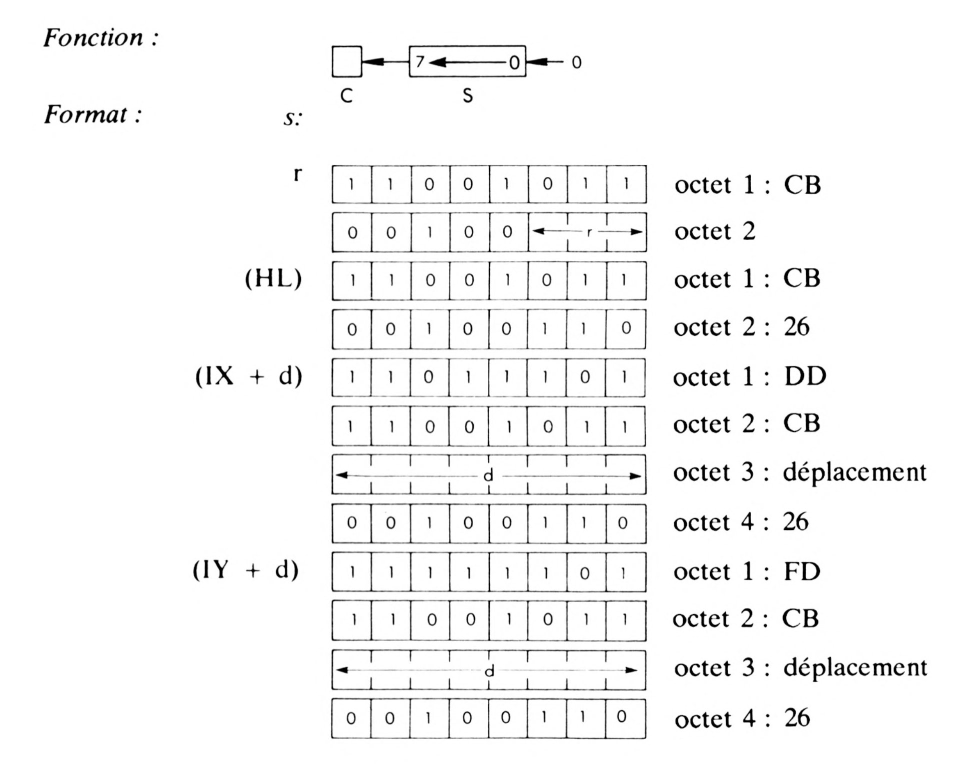
CB
FF

CODE OBJET

SLA s Décalage arithmétique à gauche de l'opérande s.

Fonction :

Format :

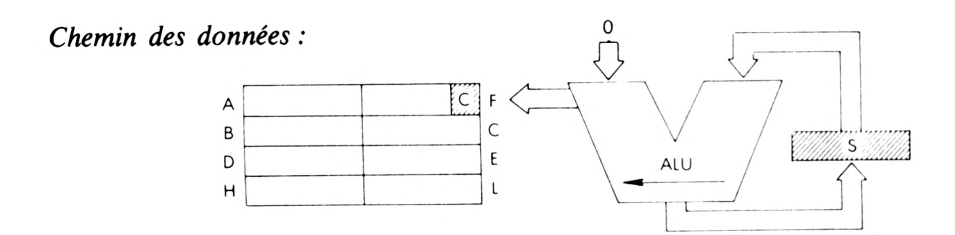


r peut être n'importe lequel de :

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Description : Le contenu de l'emplacement déterminé par l'opérande spécifié est décalé arithmétiquement vers la gauche, le contenu du bit 7 allant dans l'indicateur de report et un 0 étant forcé dans le bit 0. Le résultat final est rangé à nouveau dans l'emplacement initial. s est défini dans la description des instructions RLC similaires.

Chemin des données :

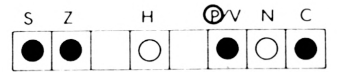


Durée :

s:	cycles M :	temps T :	usec @ 2 MHz:
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Mode d'adressage : r : implicite ; (HL) : indirect ; (IX + d), (IY + d) : indexé.

Codes : SLA r



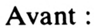
Indicateurs :

S	Z		H		P/V	N	C
●	●		○		●	○	●

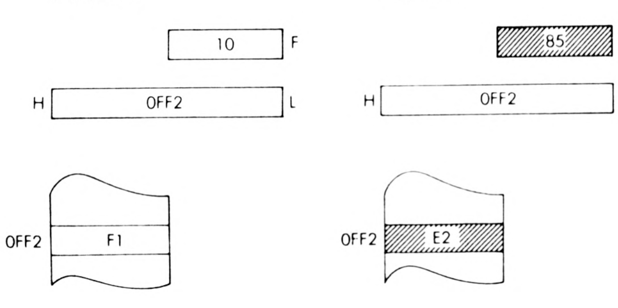
C est positionné par le bit 7 de la donnée source

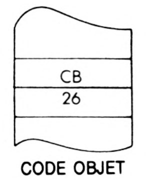
Exemple SLA (HL)

Avant : Après :





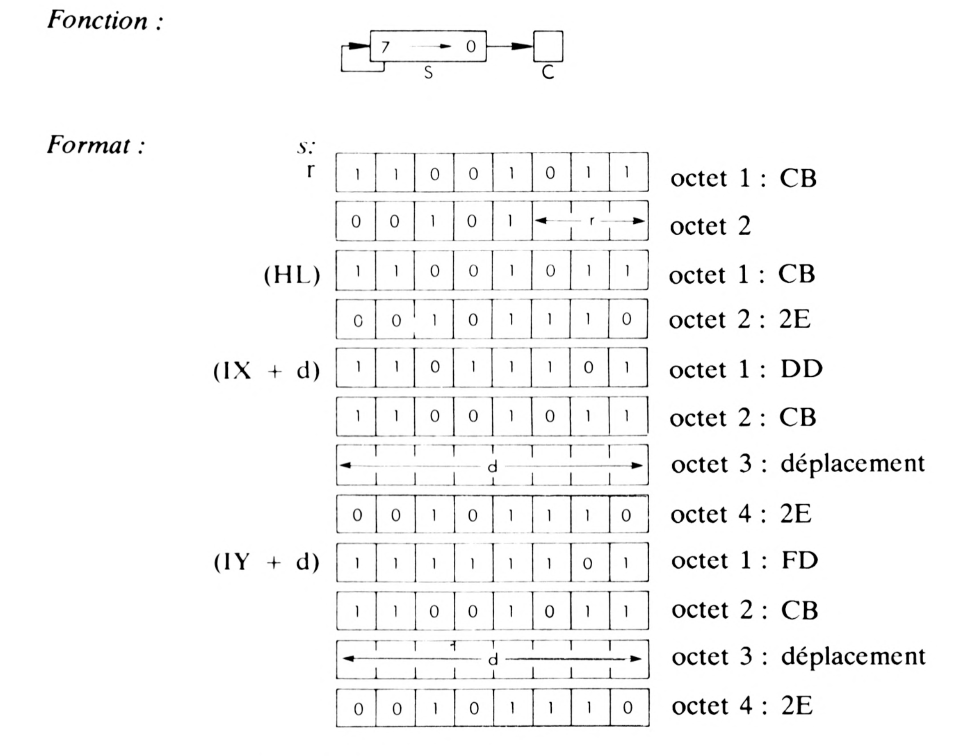




SRA s Décalage arithmétique à droite de l'opérande s.

Fonction :

Format :

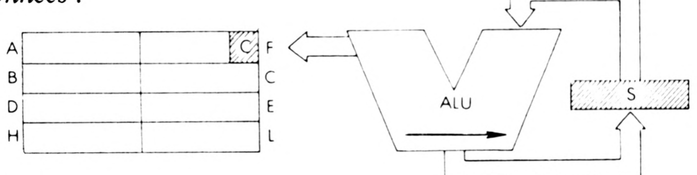


r peut être n'importe lequel de :

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Description : Le contenu de l'emplacement déterminé par l'opérande spécifié est décalé arithmétiquement vers la gauche, le contenu du bit 0 allant dans l'indicateur de report et le contenu du bit 7 demeurant inchangé. Le résultat final est rangé dans l'emplacement initial. s est défini dans la description des instructions similaires RLC.


Chemin des données :



Durée :

s:	cycles M :	temps T :	usec @ 2 MHz:
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Mode d'adressage : r : implicite ; (HL) : indirect ; (IX + d), (IY + d) : indexé.

Codes : SRA r

r:	A	B	C	D	E	H	L
CB-	2F	28	29	2A	2B	2C	2D

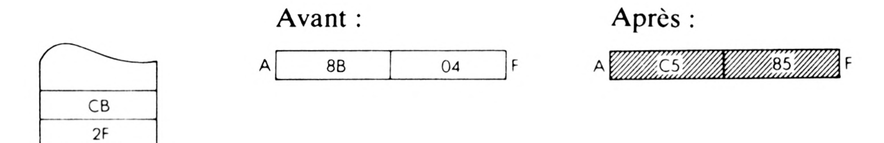
Indicateurs :

S	Z		H		P/V	N	C
●	●		○		●	○	●

C est positionné par le bit 0 de la donnée source

Exemple SRA A

Avant : Après :



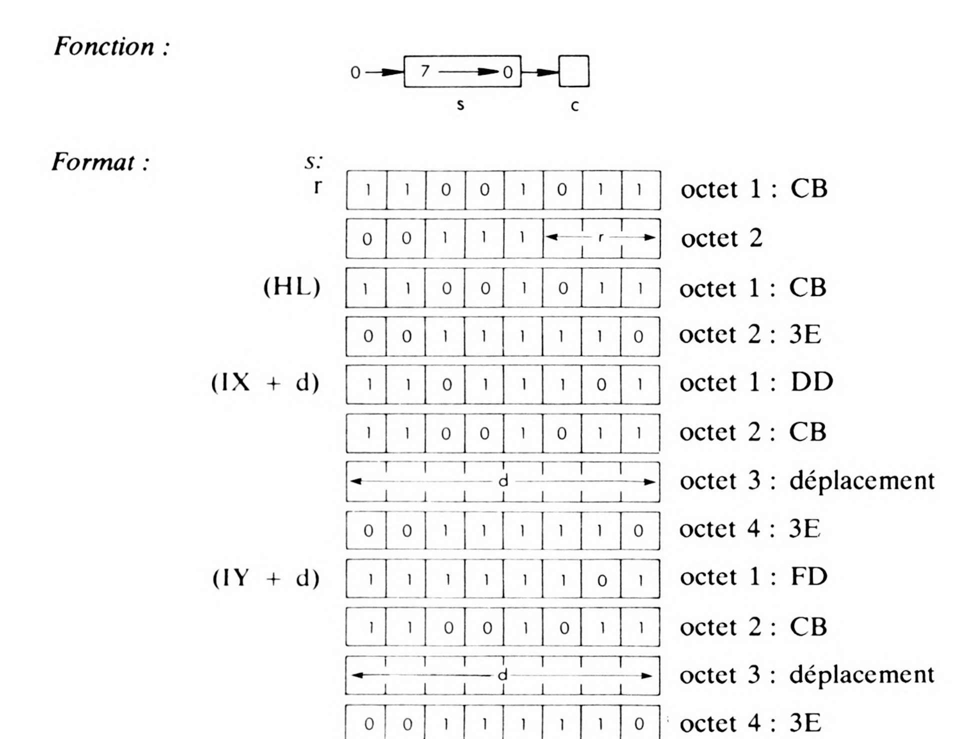
CB
2F

CODE OBJET

SRL s Décalage logique à droite de l'opérande s.

Fonction :

Format :

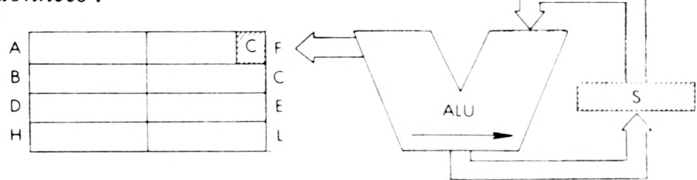


r peut être n'importe lequel de :

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Description : Le contenu de l'emplacement déterminé par l'opérande spécifié est décalé logiquement vers la droite. Un zéro est placé dans le bit 7 et le contenu du bit 0 va dans l'indicateur de report. Le résultat est rangé à nouveau dans l'emplacement initial.

Chemin des données :

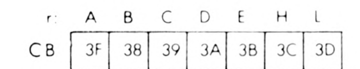


Durée :

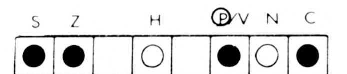
s:	*cycles M :*	*temps T :*	*usec @ 2 MHz:*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Mode d'adressage : r : implicite ; (HL) : indirect ; (IX + d), (IY + d) : indexé.

Codes : SRL r



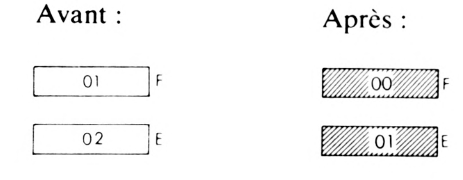
Indicateurs :



C est positionné par le bit 0 de la donnée source

Exemple SRL E

Avant : Après :



SUB A, s

Soustraire l'opérande s de l'accumulateur.

Fonction : A ← A − s

Format : s : peut être r, n, (HL), (IX + d) ou (IY + d)

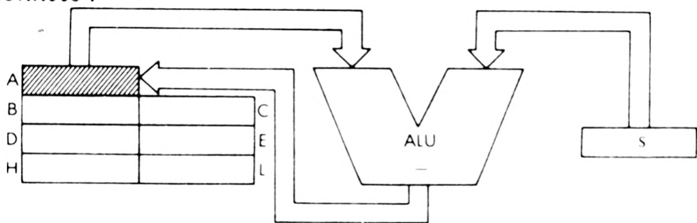
r	1	0	0	1	0	← r →			
n	1	1	0	1	0	1	1	0	octet 1 : D6
	← n →								octet 2 : valeur immédiate
(HL)	1	0	0	1	0	1	1	0	96
(IX + d)	1	1	0	1	1	1	0	1	octet 1 : DD
	1	0	0	1	0	1	1	0	octet 2 : 96
	← d →								octet 3 : déplacement
(IY + d)	1	1	1	1	1	1	0	1	octet 1 : FD
	1	0	0	1	0	1	1	0	octet 2 : 96
	← d →								octet 3 : déplacement

r peut être n'importe lequel de :

A − 111
B − 000
C − 001
D − 010
E − 011
H − 100
L − 101

Description : L'opérande spécifié s est soustrait du contenu de l'accumulateur, et le résultat est rangé dans l'accumulateur. s est défini dans la description des instructions similaires ADD.

Chemin des données :

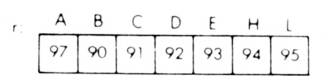


Durée :

s:	cycles M :	temps T :	usec @ 2 MHz:
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IX + d)	5	19	9.5

Mode d'adressage : r : implicite ; n : immediat ; (HL) : indirect ; (IX + d), (IY + d) : indexé

Codes : SUB r



r:	A	B	C	D	E	H	L
	97	90	91	92	93	94	95

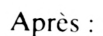
Indicateurs :

S	Z		H		P/V	N	C
●	●		●		●	1	●

Exemple SUB B

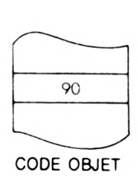
Avant :

A	80
B	31



Après :

A	4F
B	31



XOR s Ou exclusif logique entre l'accumulateur et l'opérande s.

Fonction : A ← A ⊻ s

Format : s : may be r, n, (HL), (IX + d), ou (IY + d)

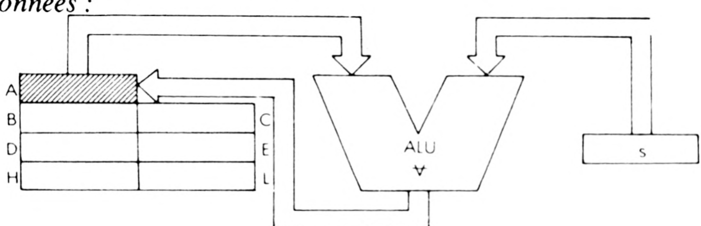
r	1 0 1 0 1 ←r→	
n	1 1 1 0 1 1 1 0	octet 1 : EE
	←n→	octet 2 : donnée immédiate
(HL)	1 0 1 0 1 1 1 0	octet 1 : AE
(IX + d)	1 1 0 1 1 1 0 1	octet 1 : DD
	1 0 1 0 1 1 1 0	octet 2 : AE
	←d→	octet 3 : déplacement
(IY + d)	1 1 1 1 1 1 0 1	octet 1 : FD
	1 0 1 0 1 1 1 0	octet 2 : AE
	←d→	octet 3 : déplacement

r may be any one of :

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Description : Le ou exclusif logique de l'accumulateur et de l'opérande s spécifié est effectué et le résultat est rangé dans l'accumulateur. s est défini dans la description des instructions ADD similaires.

Chemin des données :

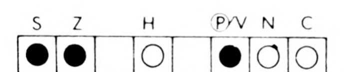


Durée :

s:	*cycles M :*	*temps T :*	*usec @ 2 MHz:*
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Mode d'adressage : r : implicite ; n : immediat ; (HL) : indirect ; (IX + d), (IY + d) : indexé

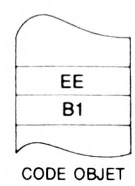
Codes : **XOR r**



Indicateurs :

S	Z		H		P/V	N	C
●	●		○		●	○	○

Exemple : **XOR, B1H**



Avant :

A 36

Après

A 87

5

TECHNIQUES D'ADRESSAGE

INTRODUCTION

Ce chapitre présente la théorie générale de l'adressage, et les différentes techniques développées pour faciliter la manipulation des données. Dans la seconde partie, nous passerons en revue les modes d'adressages spécifiques du Z80, en mettant en évidence leurs avantages et leurs limitations. Finalement, pour familiariser le lecteur aux différentes techniques, nous étudierons quelques applications particulières.

Le Z80 dispose, en plus du compteur ordinal PC, de plusieurs registres 16 bits, qui peuvent servir à désigner des adresses. Il est donc important que l'utilisateur comprenne bien les différentes techniques d'adressage, et en particulier, le rôle des registres d'index. Les modes d'adressage complexes pourront être négligés, lors d'une première lecture. Cependant, tous les modes d'adressage sont utiles au développement d'un programme. Voyons maintenant les différentes techniques.

LES MODES D'ADRESSAGE

L'*adressage* désigne la manière dont est spécifiée l'adresse de l'opérande sur lequel l'instruction va travailler. Les principales méthodes d'adressage sont toutes représentées à la figure 5.1.

Adressage implicite (ou « registre »)

Les instructions travaillant uniquement sur des registres utilisent *l'adressage implicite* [figure 5.1]. Une instruction est dite implicite lorsqu'elle ne contient pas explicitement l'adresse de l'opérande sur lequel elle travaille. A la place, son code opération désigne un ou plusieurs registres, souvent

l'accumulateur. En général, bien peu de registres internes sont disponibles (la plupart du temps 8). Il suffira donc d'un petit nombre de bits pour les désigner. Par exemple, trois bits d'une instruction suffiront à désigner un registre parmi huit possibles. De telles instructions pourront souvent être codées sur 8 bits. C'est un avantage important, puisqu'une instruction sur huit bits est, en principe, exécutée plus rapidement qu'une instruction sur deux ou trois octets.

Exemple d'une telle instruction :

LD A, B

qui signifie « transférer le contenu du registre B dans A » (charger A avec la valeur de B).

Adressage immédiat

L'adressage immédiat est illustré à la figure 5.1. Le code opération sur huit bits est suivi d'un littéral (une constante) sur 8 ou 16 bits. Ce type d'instruction est utile, par exemple, pour mettre une valeur donnée, sur 8 bits, dans un registre. Dans la mesure où le microprocesseur dispose d'instructions sur 16 bits, il sera nécessaire d'avoir recours à des littéraux eux-mêmes sur 16 bits. Exemple d'instruction immédiate :

ADD A, 0H

Le second mot de l'instruction contient le littéral « 0 », qui est additionné au contenu de l'accumulateur.

Adressage absolu

L'adressage est absolu, lorsque le code opération est suivi d'une adresse de 16 bits : celle de l'opérande sur lequel travaille l'instruction. L'adressage absolu nécessite donc des instructions de trois octets.

Exemple :

LD (1234H), A

Cela signifie : ranger le contenu de l'accumulateur dans la case mémoire d'adresse 1234 hexadécimal.

L'inconvénient de ce type d'adressage est de requérir une instruction sur trois octets. Pour améliorer l'efficacité du microprocesseur, un autre mode d'adressage est possible. Il n'exige qu'un octet supplémentaire pour désigner l'adresse : l'adressage direct.

Adressage direct (ou « court », ou « relatif »)

Dans ce mode d'adressage, le code opération est suivi d'une adresse sur huit bits [figure 5.1]. L'avantage de cette approche est de ne requérir que

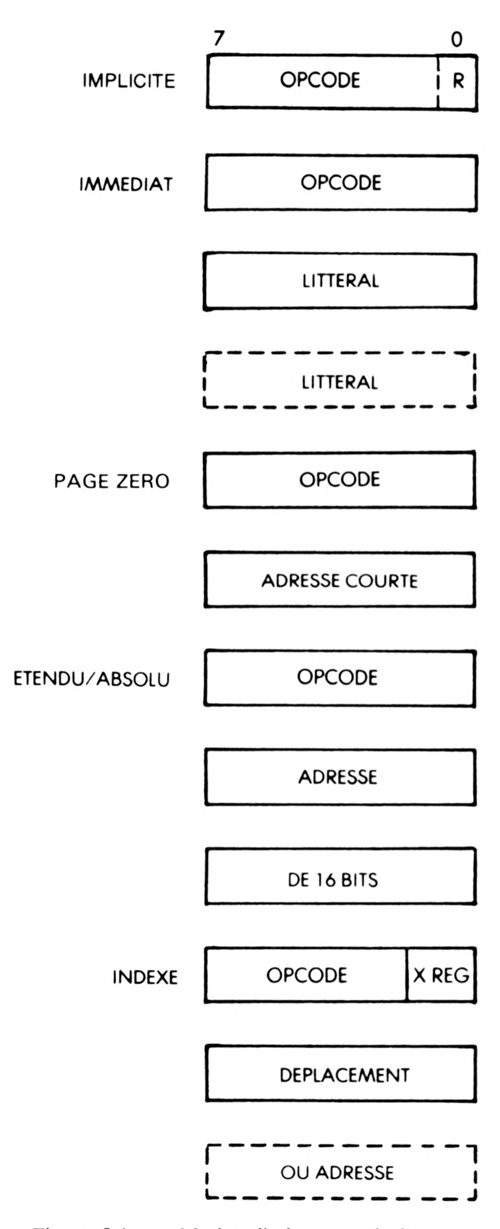


Figure 5.1. — Modes d'adressage de base

deux octets, au lieu de trois, pour l'adressage absolu. Son inconvénient est de limiter l'adressage aux adresses 0 – 255, ou bien –128 à +127. Lorsqu'il désigne les adresses 0 à 255 (« page zéro »), il est aussi appelé « adressage court », ou « adressage de la page 0 ». Quand l'adressage court est disponible, l'adressage absolu est souvent appelé, par opposition, *« adressage étendu »*. L'intervalle — 128, 127 est utilisé par les instructions de saut : il est dit « adressage relatif ».

Adressage relatif

Les instructions normales de saut, ou de branchement, demandent huit bits pour le code opération, et 16 bits supplémentaires pour préciser l'adresse où aura lieu le saut. De même que dans l'exemple précédent, l'inconvénient de ce dispositif est de nécessiter trois mots, soit trois cycles mémoire. Pour permettre des branchements plus efficaces, *l'adressage relatif* utilise des instructions sur deux mots seulement. Le premier indique qu'il s'agit d'un branchement, et précise le test à effectuer, s'il s'agit d'un branchement conditionnel. Le second indique un déplacement, qui peut être positif ou négatif. Une instruction de branchement relatif permet donc un déplacement en avant de 127 octets, ou en arrière de 128 octets, par rapport au contenu de PC. (Habituellement, l'intervalle est – 126, + 129, puisque pendant le décodage de l'instruction, le PC est augmenté de 2). Les branchements à des instructions peu éloignées sont les plus fréquents. Conséquence : les instructions de branchement relatif sont souvent employées, et elles améliorent sensiblement l'efficacité des petits programmes. Par exemple, nous avons déjà utilisé l'instruction JR NC, qui signifie « saut si pas de report » vers une adresse située à moins de 127 octets de l'instruction (plus précisément, de + 129 à – 126).

Les deux avantages de l'adressage relatif sont l'amélioration des performances du programme (moins d'octets utilisés, vitesse plus grande), et la possibilité de reloger le programme en mémoire (indépendance vis-à-vis des adresses d'implantation, absolues en l'occurrence).

Adressage indexé

L'adressage indexé est une technique permettant d'accéder successivement aux éléments d'un bloc, ou d'une table. Cette propriété sera illustrée, plus loin dans ce chapitre, par des exemples. Son principe est que l'instruction spécifie à la fois un registre d'index et une adresse. Le contenu du registre est ajouté à l'adresse, et la somme constitue l'adresse de l'opérande. De cette manière, l'adresse peut être alors celle du début d'une table en mémoire. Le registre d'index peut, lui, être utilisé pour accéder, successivement et de manière efficace, aux différents éléments de la table. (Ce qui requiert la disponibilité d'instructions d'incrément et de décrément du registre d'index). En pratique, des restrictions existent souvent, qui limitent, soit la taille du registre d'index, soit celle de l'adresse (ou déplacement).

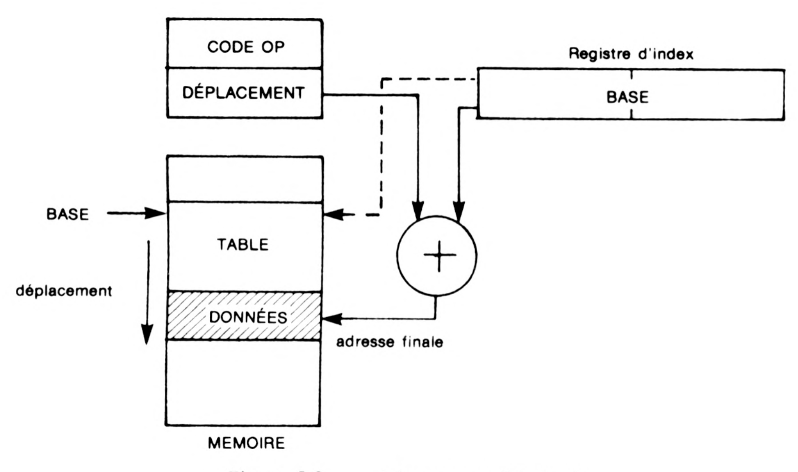


Figure 5.2. — Adressage pré-indexé

Pré-indexation et post-indexation

Deux modes d'indexation peuvent être distingués. La préindexation est le mode habituel, dans lequel l'adresse finale est la somme d'un déplacement (ou adresse) et du contenu du registre d'index [figure 5.2], en supposant un déplacement sur 8 bits et un registre d'index sur 16 bits.

La post-indexation considère que les bits ne représentent pas la valeur du déplacement, mais plutôt l'adresse de ce dernier en mémoire [figure 5.3].

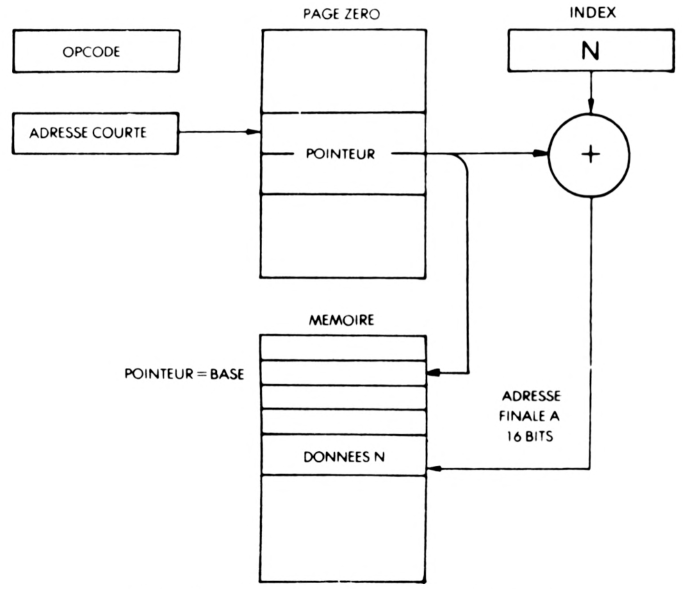


Figure 5.3. — Adressage indirect post-indexe

Dans ce mode, l'adresse finale est la somme du contenu du registre d'index et du contenu de l'adresse mémoire précisée dans *la partie déplacement du code de l'instruction.* Cette technique d'adressage utilise, en fait, une combinaison de l'adressage indirect et de l'adressage pré-indexé. Le premier n'a pas encore été défini, et nous allons maintenant nous y attacher.

Adressage indirect

On sait que deux sous-programmes peuvent éprouver le besoin d'échanger une grande quantité de données stockées en mémoire. Plus généralement, plusieurs programmes (ou sous-programmes) peuvent solliciter l'accès à un bloc commun d'information. Pour conserver un caractère de généralité au programme, il n'est pas indifférent qu'un tel bloc de données ne soit pas forcément implanté à une adresse fixe en mémoire. La taille de ce bloc peut notamment grandir ou diminuer dynamiquement, et il peut être intéressant que son implantation mémoire varie en fonction de sa taille. Il devient alors difficile, et peu pratique, d'essayer d'accéder à ce bloc de données au moyen d'adresses absolues. Les programmes devraient, en effet, être réécris à chaque fois.

La solution consiste à déposer, à une adresse fixe en mémoire, l'adresse de début du bloc considéré. Cela est tout à fait comparable à la situation où plusieurs personnes souhaitent entrer dans une maison dont il n'existe qu'une seule clé. Solution possible : cacher, en toute occurrence, la clé sous le paillasson. Chaque utilisateur sait où il doit regarder (sous le paillasson) pour trouver la clé de la maison (ou peut-être l'adresse d'un rendez-vous, pour s'en tenir à une stricte analogie avec le problème qui nous concerne). L'adressage indirect utilise un code opération (16 bits sur le Z80), suivi d'une adresse sur 16 bits, permettant de retrouver un mot en mémoire. Il s'agit, généralement, d'un mot de 16 bits (deux octets) puisqu'on a affaire à une adresse [cf. figure 5.4]. Les deux octets rangés à l'adresse A1 contiennent l'adresse A2, qui est l'adresse réelle des données auxquelles nous désirons avoir accès.

L'adressage indirect est précieux, chaque fois que des pointeurs sont utilisés. Différentes instructions du programme peuvent alors faire référence à ces pointeurs pour accéder facilement, et élégamment, à un même bloc de données. L'adresse finale peut aussi être obtenue en désignant, dans l'instruction, le registre de 16 bits qui la contient. Cela s'appelle un « adressage indirect par registre ».

Combinaisons de modes

Les modes d'adressage que nous venons de décrire peuvent être combinés. En particulier, il devrait être possible, dans une organisation d'adressage généralisé, d'utiliser plusieurs niveaux d'indirection. L'adresse A2 pourrait être à son tour considérée comme indirecte, etc.

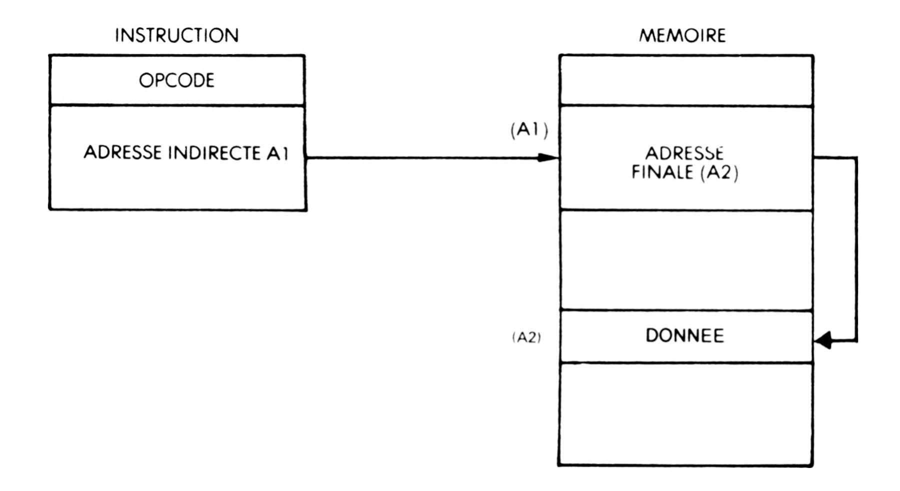


Figure 5.4. — Adressage indirect

L'adressage indexé peut aussi être combiné avec l'adressage indirect, pour permettre un accès efficace au mot d'un bloc de données, à condition de savoir où se trouve le pointeur sur le premier mot de ce bloc (voir figure 5.2).

Nous sommes maintenant plus familiers des modes d'adressage courants. La plupart des systèmes microprocesseurs, en raison de la limitation apportée à la complexité du MPU, qui doit être construit en un seul circuit, ne proposent pas tous les modes d'adressage, mais seulement un nombre limité d'entre eux. Le Z80 propose, pour sa part, une bonne partie de ceux que nous venons de voir, et que nous allons maintenant examiner.

LES MODES D'ADRESSAGE DU Z80

Adressage implicite (Z80)

L'adressage implicite est essentiellement utilisé par les instructions sur un seul octet travaillant sur des registres internes. Chaque fois que des instructions implicites travaillent exclusivement de cette façon, leur exécution ne dure pas plus d'un cycle.

Exemples de telles instructions : LD r, r' ; ADD A, r ; ADC A, s ; SUB s ; SBC A, s ; AND s ; OR s ; XOR s ; CP s ; INC r.

Zilog fait une distinction supplémentaire entre les adressages de registre et implicite. Cette définition limite le dernier nommé aux instructions qui ne possèdent pas, dans leur code, de zone réservée à la désignation d'un registre. Cela conduit à introduire un mode d'adressage supplémentaire, et c'est pourquoi le nombre de modes d'adressage ne suffit pas à caractériser les possibilités d'un microprocesseur.

Adressage immédiat (Z80)

Le Z80 disposant à la fois de registres simple longueur (8 bits), et de paires de registres double longueur (16 bits), permet deux modes d'adressage immédiat : l'un avec des littéraux de 8 bits, l'autre avec des littéraux de 16 bits. Les instructions correspondantes sont alors codées sur deux ou trois octets. Le second (huit bits) ou le second et le troisième (16 bits) contiennent la constante, ou littéral, à charger dans un registre, ou à utiliser pour une opération. Les instructions LD IX et LD IY sont des exceptions, car elles exigent des codes opération sur 16 bits ; la constante littérale occupe alors le troisième octet.

Exemples d'instruction utilisant l'adressage immédiat :

LD r, n (2 octets)
LD dd, nn (3 octets)

et

ADD A, n (2 octets)

Lorsque, dans le cas du Z80, le littéral tient sur 16 bits, l'adressage est appelé : « adressage immédiat étendu ».

Adressage absolu ou « étendu » (Z80)

Par définition, l'adressage absolu requiert trois octets. Le premier est celui du code opération, et les deux autres une adresse sur 16 bits (l'adresse absolue).

Par opposition avec l'adressage court (adresse sur 8 bits), ce mode est aussi appelé « adressage étendu ».

Exemples d'instructions utilisant l'adressage étendu :

LD HL, (nn) et JP nn

où nn représente une adresse mémoire sur 16 bits, et (nn) son contenu.

Adressage page zéro modifié (Z80)

L'adressage de la page zéro n'est pas disponible sur le Z80, sauf au travers des instructions RST. Le mode spécial d'adressage utilisé par ces instructions est appelé « adressage page zéro modifié ».

Les instructions RST contiennent un champ de trois bits ($b_5b_4b_3$) servant à désigner l'une de huit adresses mémoires en page zéro. L'adresse chargée dans le PC est $b_5b_4b_3000$. Ces instructions sont exécutées rapidement, car elles ne demandent qu'un seul octet, et sont facilement exécutées par le hardware. Elles sont généralement utilisées pour répondre à des sources d'interruption multiples (jusqu'à 8). Leur inconvénient est, soit de limiter la

séquence à exécuter à 8 octets, soit de demander la réalisation d'un saut, ce qui atténue l'avantage de la rapidité. Cela tient au fait que chacune des adresses de branchement est éloignée de l'autre de 8 octets.

Cette instruction est utilisée moins souvent, depuis l'apparition des contrôleurs de priorité d'interruption (PIC) (voir nos livres C 201, ou C 207, pour une description détaillée des PIC's). Un PIC génère automatiquement une instruction de saut sur trois octets, en réponse à une interruption.

Aujourd'hui, cette instruction est généralement utilisée pour effectuer des redémarrages.

Adressage relatif (Z80)

Par définition, l'adressage relatif demande deux octets. Le premier est le code opération de « saut relatif », le second représente le déplacement et son signe.

Pour différencier ce mode du saut absolu, on le note : JR.

La durée d'exécution de l'instruction mérite que nous y portions attention. Lorsque le test échoue, autrement dit lorsqu'il n'y a pas de branchement, elle ne requiert que sept cycles d'horloge (T), parce que l'adresse de la prochaine instruction exécutable se trouve déjà dans le compteur ordinal.

Lorsque le test réussit, et que le branchement doit avoir lieu, elle demande 12 cycles d'horloge ; une nouvelle adresse doit, en effet, être calculée et chargée dans le compteur ordinal.

Le calcul de la durée d'exécution d'un morceau de programme nécessite certaines précautions. S'il n'est pas certain que le saut doive se produire, il faudra prendre en considération le fait que l'instruction peut tout aussi bien demander 12 cycles d'horloge (condition remplie), que 7 (condition non remplie).

L'exécution d'une boucle éventuelle sera plus rapide en utilisant un test, qui ne sera, lui, généralement pas vérifié. Par exemple, une condition de non-nullité d'un compteur.

Lorsque des JR sont utilisés ailleurs que dans des boucles, et que la condition à tester est inconnue, une valeur moyenne sera choisie pour le temps d'exécution du JR.

Ce problème de durée n'existe pas pour le saut inconditionnel JR e, qui ne teste aucune condition, et dure toujours 12 cycles d'horloge.

Adressage indexé (Z80)

Ce mode d'adressage n'existait pas sur le 8080 : il a été ajouté sur le Z80, en même temps que les deux registres d'index. Il a, par conséquent, été nécessaire d'ajouter un octet supplémentaire au code opération, pour les instructions utilisant les registres d'index. Le Z80 dispose donc d'instructions dont le code opération tient sur 16 bits (LDIR en est un autre exemple). La structure d'une instruction indexée apparaît à la figure 5.5.

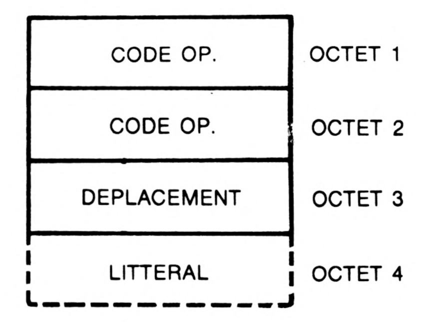


Figure 5.5. — L'adressage indexé, deux octets de code opération

Parmi les instructions permettant l'usage de l'adressage indexé se trouvent :

LD, ADD, INC, RLC, BIT, SET

Ce mode d'adressage sera largement utilisé par les programmes travaillant sur des blocs de données, des tables ou des listes.

Adressage indirect (Z80)

Le Z80 fournit un adressage indirect limité, appelé « adressage indirect par registre ». Dans ce mode, chaque paire [BC, DE et HL] peut être utilisée en tant qu'adresse mémoire.

Lorsqu'elles pointent sur des données de 16 bits, c'est toujours vers leur partie basse. La partie haute se trouve en mémoire à l'adresse immédiatement supérieure.

Combinaisons de modes

Les combinaisons de mode n'existent pas, sauf pour les instructions travaillant sur deux opérandes, dont chacun utilise un mode différent.

Ainsi, une instruction de chargement (LD), ou une instruction arithmétique, peut avoir accès de manière immédiate à l'un des opérandes, et de manière indexée à l'autre.

De même, le mécanisme d'adressage d'un bit peut adresser l'octet contenant ce bit selon l'un ou l'autre des trois modes (voir chapitre suivant).

Les modes d'adressage spécifiques à chaque instruction sont également précisés dans les tables du chapitre précédent.

Adressage de bits

L'adressage des bits n'est généralement pas considéré comme un mode d'adressage, si l'adressage élémentaire de la mémoire se fait au niveau de *l'octet*. Cependant, qu'il soit défini en tant que mode d'adressage ou que groupe d'instructions, il offre des possibilités fort intéressantes. Dans la mesure où il est décrit en tant que mode d'adressage dans la documentation Zilog, nous ferons de même ici. Ce mode est spécifique au Z80, et n'existe pas sur le 8080.

L'adressage de bits désigne le mécanisme d'accès aux bits dont on sollicite l'usage. Le Z80 fournit des instructions permettant de positionner et de tester des bits situés en mémoire, ou dans les registres. Trois modes différents permettent d'accéder à l'octet contenant le bit considéré : registre, indirect-registre et indexé. Trois bits du code opération spécifient le bit à traiter, parmi les huit de l'octet concerné.

UTILISATION DES MODES D'ADRESSAGE DU Z80

Adressages long et court

Nous avons déjà utilisé des instructions de saut relatif, dans certains programmes de petites dimensions. Leur compréhension est immédiate. Une question intéressante se pose : que pouvons-nous faire lorsque l'intervalle disponible pour un saut relatif est trop petit pour nos besoins ? Solution très simple : utiliser un *saut long*. Il s'agit d'une instruction de saut contenant l'adresse absolue (« longue »), où il est nécessaire de se brancher.

```
JR NC, $ + 3        se brancher à l'adresse courante + 3 si l'indica-
                    teur de report vaut 0
JP FAR              sinon, aller à FAR
```

Le programme précédent a pour résultat le branchement à l'adresse FAR, chaque fois que l'indicateur de report est positionné. Cela résout notre problème de saut long. Envisageons maintenant quelques exemples de modes d'adressage plus complexes. A savoir, l'indexation et l'indirection.

Utilisation de l'indexation pour l'accès à un bloc séquentiel

L'indexation est principalement utilisée pour adresser des emplacements consécutifs dans une table. La seule restriction est que leur longueur soit inférieure à 256, de manière que le déplacement puisse tenir dans un seul octet.

La recherche d'un caractère n'a plus de secrets pour nous. Nous allons donc écrire un petit programme, qui parcourra une table de cent éléments, à

la recherche du caractère « * ». L'adresse du début de cette table sera appelée BASE. Voici le programme (voir organigramme figure 5.6) :

```
RECHERCHE  LD   IX, BASE
           LD   A, '*'
           LD   B, COMPTE — D — OCTETS
TEST       CP   (IX)
           JR   Z, TROUVE
           INC  IX
           DEC  B
           JR   NZ, TEST
PAS-TROUVE ....
```

Une version améliorée de ce programme sera présentée ultérieurement, dans le paragraphe consacré aux transferts de blocs.

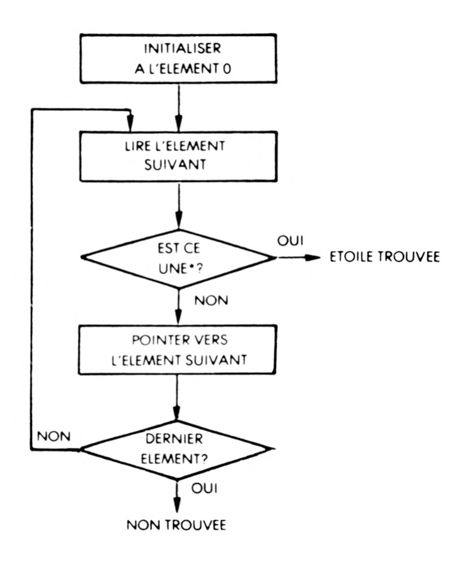


Figure 5.6. — Ordinogramme de la recherche de caractères

Un programme de transfert de blocs pour moins de 256 éléments

Nous appellerons COMPTE le nombre d'éléments du bloc à déplacer. Ce nombre sera supposé inférieur à 256. DEPART sera l'adresse de début du bloc, et ARRIVEE l'adresse du début de la zone vers laquelle le bloc sera

déplacé. L'algorithme est très simple : un mot sera déplacé à chaque fois, en conservant son numéro dans le registre C. Voici le programme :

```
DEPLACER  LD    IX, DEPART
          LD    IY, ARRIVEE
          LD    C, COMPTE
SUIVANT   LD    A, (IX)      ; ACQUERIR LE MOT
          LD    (IY), A
          INC   IX
          INC   IY
          DEC   C
          JR    NZ, SUIVANT
```

Examinons ce programme pas à pas :

```
DEPLACER  LD    IX, DEPART
          LD    IY, ARRIVEE
          LD    C, COMPTE
```

Ces trois instructions initialisent, respectivement, les registres IX, IY et C (cf. figure 5.7). Le registre d'index IX sert de pointeur source, et sera incrémenté régulièrement ; IY est le pointeur destination. Il sera, lui aussi, incrémenté régulièrement. C est chargé avec le nombre total de mots à déplacer [maximum 256], et sera décrémenté régulièrement. Lorsque C

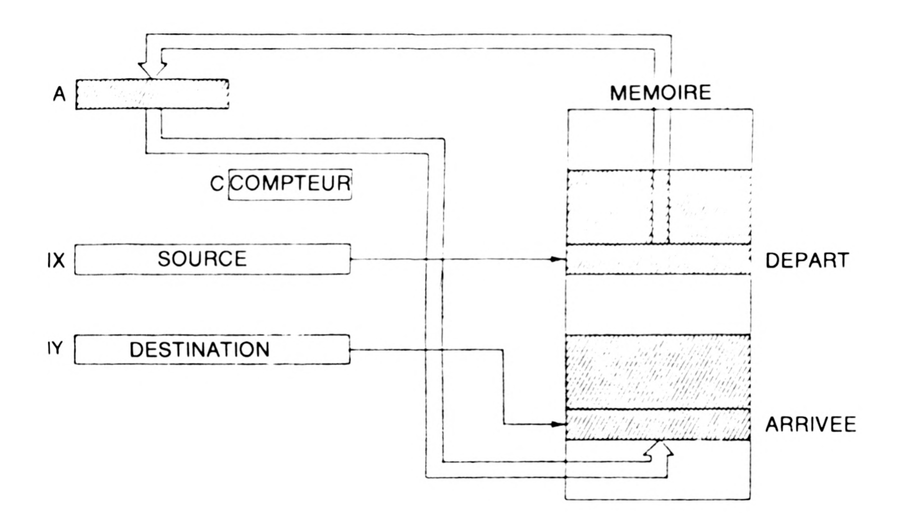


Figure 5.7. — Transfert de bloc : initialisation des registres.

atteindra zéro, tous les éléments auront été transférés. Les deux instructions suivantes,

```
SUIVANT   LD    A, (IX)
          LD    (IY), A
```

chargent dans l'accumulateur le contenu de la case mémoire pointée par IX, puis le transfèrent dans la case mémoire pointée par IY. En d'autres termes, ces deux instructions transfèrent un élément du bloc-source vers le bloc-destination. Les deux registres d'index sont alors incrémentés :

```
          INC   IX
          INC   IY
```

et le compteur est décrémenté :

```
          DEC   C
```

Pour finir, tant que le compteur n'atteint pas la valeur 0, le programme reboucle à l'adresse SUIVANT :

```
          JR    NZ, SUIVANT
```

C'est un bon exemple d'utilisation des registres d'index. Comparons-le au même programme écrit pour un autre processeur, le 6502 (technologie MOS), qui possède aussi l'adressage indexé, mais utilise des conventions différentes (qui dispose autrement dit de limitations différentes par rapport au mode d'adressage indexé le plus général).

Le programme serait :

```
        LDX   # COMPTE
SUIVANT LDA   DEPART, X
        STA   ARRIVEE, X
        DEX
        BNE   SUIVANT
```

Sans entrer dans le détail, le lecteur peut immédiatement remarquer que ce programme est bien plus court que le précédent. Pour quelle raison ? Tout simplement parce que le registre d'index X est utilisé en tant que déplacement variable, alors que DEPART et ARRIVÉE le sont comme adresses fixes des zones source et destination.

Tout cela montre que, même si en théorie l'indexation est un outil puissant, elle ne conduit pas forcément à une programmation efficace, en raison des limitations que chaque microprocesseur y apporte. Une véritable indexation impliquerait la possibilité d'un déplacement sur 16 bits, en même temps qu'un registre d'index de 16 bits.

Cependant, il convient de remarquer que ce problème de déplacement de blocs est résolu, sur le Z80, par la présence d'instructions spécialisées.

Décrivons maintenant un transfert de bloc généralisé, qui pourra être écrit avec seulement quatre instructions. Pour être justes envers le Z80, nous suggérons au lecteur deux exercices supplémentaires :

Exercice 5.1 : *Ecrire un programme de transfert de blocs pour le Z80 dans le style du programme établi pour le 6502, en supposant que le registre d'index contienne un déplacement. Nous supposerons que les zones source et destination sont, toutes deux, dans la page 0 (de 0 à 256), et que, naturellement, le nombre d'éléments des zones est suffisamment petit pour empêcher qu'elles se recouvrent.*

Exercice 5.2 : *Supposons maintenant que les deux zones sont situées n'importe où en mémoire, mais qu'elles restent, toutes deux, dans la même page. Réécrire le programme ci-dessus dans ce cas.*

Programme général de transfert de blocs (plus de 256 éléments)

L'utilisation des registres et de la mémoire est présentée à la figure 5.8. Voici le programme :

```
LD BC, COMPTE          NOMBRE D'OCTETS
LD DE, DESTINATION     ADRESSE DE LA ZONE DESTINATION
LD HL, SOURCE
LDIR                   ADRESSE DE LA ZONE SOURCE
```

Mémoire utilisée : 11 octets
Temps : 21 cycles d'horloge/octet.

La première instruction est :

```
LD      BC, COMPTE
```

Elle charge, dans la paire de registres BC, le nombre d'éléments à transférer (c'est-à-dire une valeur sur 16 bits). Les deux instructions suivantes initialisent, respectivement, les paires de registres DE et HL :

```
LD      DE, DESTINATION
LD      HL, SOURCE
```

Enfin, la quatrième instruction :

```
LDIR
```

exécute le transfert désiré.

LDIR est une instruction de *transfert automatique de blocs.* L'exemple précédent en démontre, de manière évidente, la puissance. Son cycle d'exécution est le suivant :

- le contenu de la case mémoire pointée par HL est transféré dans la case mémoire pointée par DE : (DE) ← (HL).
- puis, DE est incrémenté : DE ← DE + 1.
- puis, HL est incrémenté : HL ← HL + 1.
- puis, BC est décrémenté : BC ← BC − 1.
- enfin, si après avoir été décrémenté, BC a la valeur 0, l'instruction est terminée.
  Au contraire, si BC n'est toujours pas nul, le cycle recommence.

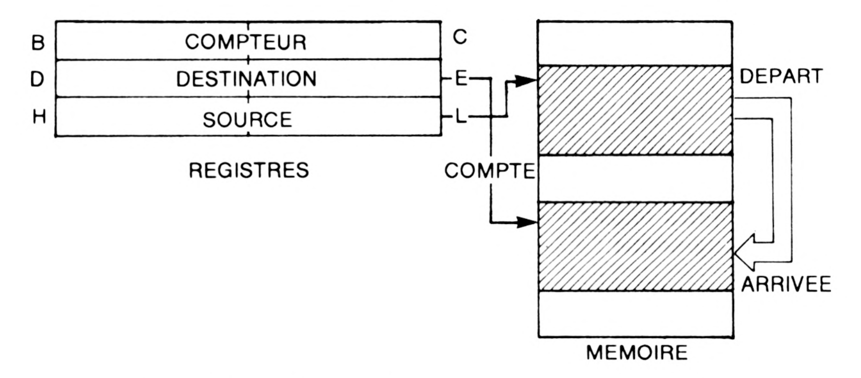


Figure 5.8. — Transfert de bloc : utilisation de la mémoire

La valeur et la puissance de l'instruction LDIR sont maintenant tellement bien établies, qu'il est inutile de les commenter davantage. De la même manière, notre recherche du caractère « étoile » dans une zone de mémoire peut être améliorée par l'utilisation d'une instruction automatique spécialisée, CPIR, qui n'existait pas sur le 8080.

Voici le programme correspondant :

```
              LD A, '*'
              LD BC, COMPTE
              LD HL, CHAINE
RECHERCHE     CPIR
              JR Z, ETOILE
PAS TROUVE    ...
```

La première instruction charge l'accumulateur avec le code du caractère étoile. La paire BC est, ensuite, initialisée avec le nombre d'octets dont dispose le bloc sur lequel la recherche va porter.

```
              LD BC, COMPTE
```

La paire de registres HL prend la valeur de la première adresse du bloc sur lequel va s'effectuer la recherche : CHAINE. L'instruction de comparaison automatique est exécutée :

```
LD HL, CHAINE
CPIR
```

L'instruction CPIR est une instruction de comparaison automatique. Le contenu de la case mémoire pointée par HL est comparé au contenu de l'accumulateur. Si la comparaison est satisfaite, autrement dit si : A = (HL), l'indicateur Z est positionné à 1. La paire de registres HL est incrémentée, la paire BC décrémentée et l'instruction est répétée jusqu'à ce que la paire BC atteigne la valeur 0, ou que la comparaison réussisse. A la fin de l'instruction CPIR, l'état de l'indicateur Z est testé, pour savoir si la comparaison a réussi (à la limite, l'instruction pourrait boucler sur les 64 K de la mémoire, sans jamais réussir la comparaison si le caractère recherché ne s'y trouve pas !). C'est la raison de la présence de la dernière instruction de notre programme

```
JR Z, ETOILE
```

Exercice 5.3 : *Réécrire le programme précédent pour effectuer la recherche en sens inverse (truc : utiliser l'instruction CPDR). « Continuer le transfert du bloc jusqu'à ce qu'une « étoile » soit trouvée ».*

Nous développerons maintenant un programme combinant les caractéristiques des deux précédents. Les contenus des cases mémoires sont transférés de la zone SOURCE vers la zone ARRIVEE. L'opération s'arrête automatiquement, lors de la rencontre d'un caractère spécial. Par exemple le caractère étoile.

Voici le programme :

```
     LD   BC, COMPTE
     LD   HL, SOURCE
     LD   DE, DESTINATION
     LD   A, '*'          DELIMITEUR (caractère de fin)
TEST CP   (HL)            COMPARER AVEC LE
     JR   Z, FIN          CONTENU MEMOIRE TERMI-
                          NER SI PAREIL
     LDI                  TRANSFERER CARACTERE,
                          METTRE LES POINTEURS ET
                          LE COMPTEUR A JOUR
     JP   PE, TEST        CONTINUER DE TESTER TANT
                          QUE P N'INDIQUE PAS QUE
                          BC = 0
```

Les trois premières instructions accomplissent les initialisations habituelles : celles du registre de comptage (BC) et des pointeurs (HL et DE) :

```
LD      BC, COMPTE
LD      HL, SOURCE
LD      DE, DESTINATION
```

Le caractère étoile est chargé, « comme d'habitude », dans l'accumulateur, de manière à être comparé aux caractères placés dans les cases mémoires.

```
LD      A, '*'
```

L'instruction suivante accomplit la comparaison :

```
TEST    CP (HL)
```

Le succès, où l'échec, de la comparaison est établi en testant l'indicateur Z. Ce dernier est positionné en cas de réussite. Le test est effectué par l'instruction suivante :

```
JR      Z, FIN
```

L'instruction qui suit est une instruction de *transfert automatique :*

```
LDI
```

Elle transfère le caractère de la zone source vers la zone destination, et met à jour les pointeurs et le compte d'octets. Le tout en une seule instruction.

Nous aurons donc successivement :

- (DE) ← (HL)
- DE ← DE + 1
- HL ← HL + 1
- BC ← BC − 1

La particularité de cette instruction est que l'indicateur P/V est mis à 1, si BC passe à 0, et laissé à 0 dans le cas contraire. Cet indicateur sera testé explicitement par la dernière instruction, pour savoir si BC est passé à 0, c'est-à-dire, pour savoir si notre opération est terminée ou non :

```
JP      PE, TEST
```

Addition de deux blocs

Nous allons maintenant écrire un programme capable d'additionner, élément par élément, les contenus de deux blocs commençant respective-

ment aux adresses BLOC1 et BLOC2, et disposant, tous deux, du même nombre d'éléments, COMPTE.

Le programme est listé ci-dessous :

```
BLOC-ADD   LD    IX, BLOC1
           LD    IY, BLOC2
           LD    B, COMPTE
           XOR   A
BOUCLE     LD    A, (IX + 0)
           ADC   A, (IY + 0)
           LD    (IX + 0), A
           DEC   IX
           DEC   IY
           DEC   B
           JR    NZ, BOUCLE
```

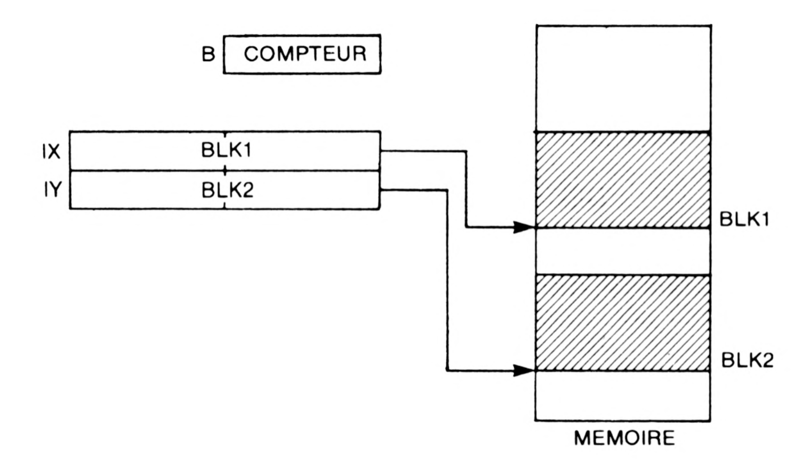


Figure 5.9. — Addition de deux blocs BLK1 = BLK1 + BLK2

L'utilisation de la mémoire est exposée à la figure 5.9. Le cheminement du programme est évident. Le nombre d'éléments à additionner est chargé dans le registre de comptage B, et les deux registres d'index IX et IY initialisés avec les valeurs BLOC1 et BLOC2.

```
BLOC-ADD   LD    IX, BLOC1
           LD    IY, BLOC2
           LD    B, COMPTE
```

L'indicateur de report (carry) est mis à 0 avant la première addition :

```
                XOR     A
```

Le premier élément est chargé dans l'accumulateur :

```
BOUCLE          LD      A, (IX + 0)
```

L'élément correspondant de BLOC2 lui est ajouté :

```
                ADC     A, (IY + 0)
```

et le résultat est finalement rangé dans le BLOC1 :

```
                LD      (IX + 0), A
```

Les deux pointeurs sont alors décrémentés :

```
                DEC     IX
                DEC     IY
```

ainsi que le compteur :

```
                DEC     B
```

Aussi longtemps que le registre de comptage n'a pas atteint la valeur 0, la boucle d'addition est exécutée :

```
                JR      NZ, BOUCLE
```

Exercice 5.4 : *Le programme ci-dessus permet-il d'effectuer l'addition de deux nombres de 32 bits ?*

Exercice 5.5 : *Le programme ci-dessus permet-il d'effectuer l'addition de deux nombres sur 64 bits ?*

Exercice 5.6 : *Modifier le programme ci-dessus de façon que le résultat soit rangé dans un troisième bloc BLOC3.*

Exercice 5.7 : *Modifier le programme ci-dessus de telle façon qu'une soustraction soit effectuée à la place d'une addition.*

Exercice 5.8 : *Modifier le programme original de manière que BLOC1 et BLOC2 soient les adresses de début de chacun des blocs, et non les adresses de fin (voir figure 5.10).*

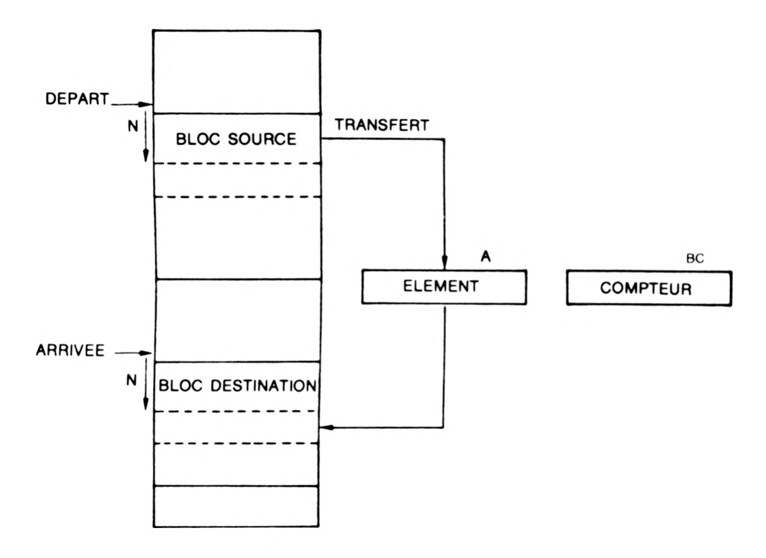


Figure 5.10. — Organisation mémoire pour transfert de bloc

CONCLUSION

Une description complète des modes d'adressage vient d'être présentée. Nous avons vu que le Z80 offre de nombreux mécanismes d'adressage. Ses modes d'adressage spécifiques ont été étudiés. Enfin, plusieurs programmes d'application ont été présentés, qui montrent l'intérêt des divers mécanismes.

La programmation du Z80 requiert, pour être efficace, une bonne compréhension de ces mécanismes, qui seront, d'ailleurs, mis à contribution tout au long des chapitres suivants.

EXERCICES

5.9 : *Ecrire un programme additionnant les dix premiers octets d'une table rangée à l'adresse «BASE». Le résultat aura 16 bits (ce type de programme est utilisé dans les calculs de mots de contrôle (en anglais : checksum).*

5.10 : *Le même problème peut-il être résolu en utilisant l'adressage indexé ?*

5.11 : *Inversez l'ordre des dix octets de la table. Rangez le résultat à l'adresse « INVERSE ».*

5.12 : *Recherchez le plus grand élément de la table, et rangez-le à l'adresse mémoire « PLUS-GRAND ».*

5.13 : *Additionner les éléments correspondants de trois tables, dont les adresses sont BASE1, BASE2 et BASE3. La longueur de ces tables est rangée, en page zéro, à l'adresse « LONGUEUR ».*

6

TECHNIQUES D'ENTRÉES/SORTIES

INTRODUCTION

Nous avons, jusqu'ici, appris à échanger des informations entre la mémoire et les différents registres du processeur, à gérer les registres et à utiliser les différentes instructions pour manipuler les données. Nous allons maintenant apprendre à communiquer avec le monde extérieur. Cela s'appelle effectuer des entrées-sorties.

On appelle *entrée* l'action d'amener des données dans le système (MPU et mémoire), depuis des périphériques extérieurs (clavier, disque, capteur...). On appelle *sortie* l'action de transférer des données depuis le microprocesseur (ou la mémoire) jusqu'à des périphériques extérieurs, tels que imprimante, écran CRT, disque, relais ou autres.

Nous procéderons en deux étapes. Nous nous efforcerons, d'abord, d'effectuer les entrées/sorties utilisées par les phériphériques courants. Ensuite, nous apprendrons à gérer simultanément plusieurs périphériques, c'est-à-dire, à planifier leur travail. La seconde partie couvrira notamment les méthodes d'interrogation et d'interruptions.

ENTREES/SORTIES

Dans ce paragraphe, nous allons essayer d'acquérir, ou de générer, des signaux simples, tels que des impulsions, et d'apprendre ensuite les techniques de mesure correcte de la durée des signaux. Nous serons alors prêts pour l'étude d'échanges plus complexes, tels que les entrées-sorties rapides, parallèles, ou en série.

Les instructions d'entrée-sortie du Z80

Le Z80 comprend un ensemble séparé d'instructions d'entrée-sortie, à l'inverse de la plupart des microprocesseurs 8 bits, qui utilisent l'ensemble des instructions pour gérer les périphériques. Le Z80, comme le 8080, dispose des instructions de base d'entrées/sorties, et en outre, d'instructions d'E/S (entrées-sorties) supplémentaires. Elles seront décrites, ici, en détail, pour faciliter la compréhension des programmes présentés.

Les instructions de base d'entrée-sortie sont, respectivement : IN A, (n) et OUT (n), A. Elles sont héritées du 8080, et effectuent la lecture et l'écriture d'un octet depuis (resp. vers) le périphérique sélecté vers (resp. depuis) l'accumulateur. Le dispositif d'adressage physique est tel que l'adresse « n » du périphérique d'E/S est présentée sur les bits A0 à A7 du bus d'adresses, alors que le contenu de l'accumulateur apparaît sur les bits A8 et A15. Dans le cas où moins de 256 périphériques sont utilisés, il pourra être nécessaire de mettre explicitement l'accumulateur à 0, si l'une des lignes A8 à A15 peut être utilisée par le dispositif de décodage d'adresse des périphériques. Dans les exemples simples que nous présenterons, nous supposerons toujours que le nombre de périphériques disponibles est inférieur à 256, de façon à éviter de mettre explicitement le contenu de l'accumulateur à 0, par exemple, avant d'utiliser l'instruction IN.

Une instruction spéciale d'entrée : IN r, (C) rend possible l'utilisation du registre C comme adresse du périphérique. Le contenu du registre B fournit alors, automatiquement, la partie haute de l'adresse (A8 à A15). Le registre précisé r reçoit les données envoyées par le périphérique ainsi adressé. « r » peut être l'un quelconque des sept registres 8 bits d'usage général.

Générer un signal

Le cas le plus simple consiste à activer, ou à désactiver, un périphérique (le mettre « on » ou « off »). Pour changer l'état de ce périphérique, le programmeur doit simplement faire passer un niveau de l'état « 0 » à l'état « 1 », ou de l'état « 1 » à l'état « 0 ». Supposons que nous disposions d'un relai connecté au bit « 0 » d'un registre nommé « OUT1 ». Pour le mettre « on » (pour le faire « coller »), nous aurons simplement à écrire un « 1 » dans le bit concerné du registre OUT1. Nous supposerons que OUT1 représente l'adresse de ce registre pour notre système. Un programme capable de mettre « on » le relai est, par exemple :

```
METTRE-ON    LD     A,00000001 B    CHARGER LA VALEUR 1
             OUT    (OUT1),A        DANS A
```

dans lequel OUT est l'instruction de sortie.

Nous avons postulé, a priori, que l'état des sept autres bits importe peu. Ce n'est cependant pas souvent le cas. Ces bits auraient pu être connectés à d'autres relais. Efforçons-nous d'améliorer ce programme. L'objectif est de fermer le relai (le mettre « on ») sans changer l'état d'aucun autre bit du

registre. Nous allons supposer que les opérations de lecture sont possibles, sur ce registre.

Notre programme devient :

```
METTRE-ON   IN    A, (OUT1)    LIT LE CONTENU DE OUT1
            OR    00000001B    FORCE LE BIT 0 A « 1 »
            OUT   (OUT1), A
```

Le programme commence par lire le contenu de OUT1, puis il effectue un OU sur son contenu, qui ne change que le bit 0 et laisse les autres intacts. (Pour plus de détails sur l'instruction OR, revoir le chapitre 4). L'opération est montrée sur la figure 6.1.

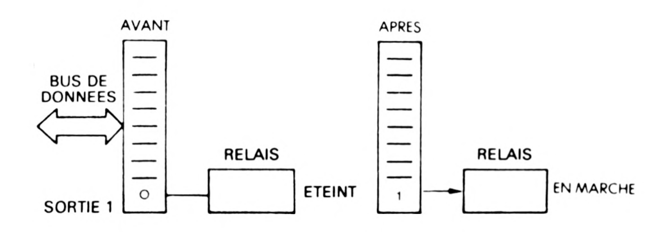


Figure 6.1. — Activer un relai

Les impulsions

La génération d'une *impulsion* s'effectue exactement de la même façon que dans le cas précédent du *niveau.* Une sortie est d'abord mise « on », et ensuite « off » (cf. figure 6.2). Cette fois cependant, un problème supplémentaire doit être résolu : celui de la durée de l'impulsion. Aussi allons-nous étudier la génération d'un délai connu.

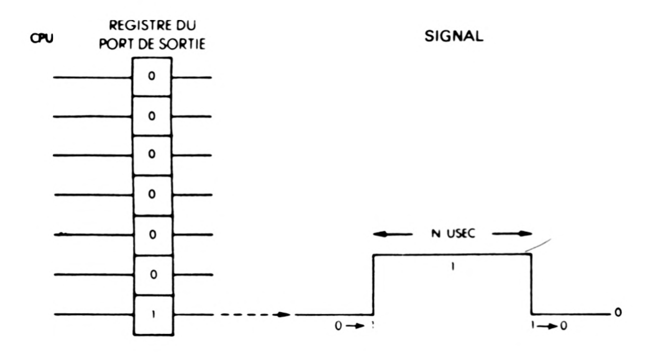


Figure 6.2. — Une impulsion programmée

Génération et mesure de délais

Un délai peut être généré, soit par programme, soit au moyen de hardware. Nous étudierons, ici la génération par programme, et envisagerons ultérieurement la possibilité de la mettre en œuvre au moyen d'un compteur spécial, appelé générateur d'intervalles programmable.

Générons des délais au moyen d'un comptage. Un registre de comptage est d'abord chargé avec une certaine valeur, puis décrémenté. Le programme boucle sur lui-même, et continue de décrémenter jusqu'à ce que le compteur atteigne la valeur « 0 ».

Le temps total utilisé par cette boucle de programme nous servira de délai.

Nous générerons, par exemple, un délai de 82 cycles d'horloge :

```
DELAI     LD    A, 5          A SERT DE COMPTEUR
BOUCLE    DEC   A
          JR    NZ, BOUCLE
```

Ce programme charge A avec la valeur 5. L'instruction suivante décrémente A, et la dernière provoque un branchement à BOUCLE, tant que A n'atteint pas la valeur « 0 ». Lorsqu'enfin il l'atteint, le programme sort de la boucle et exécute les instructions suivantes. La logique du programme est simple ; elle apparaît sur l'organigramme de la figure 6.3.

Calculons maintenant le délai généré par ce programme. Dans le chapitre quatre de ce livre, nous retrouverons le nombre de cycles requis par l'exécution de chaque instruction.

LD A dans le mode immédiat, demande sept cycles d'horloge. DEC utilise quatre cycles. Finalement, JR utilise douze cycles, sauf pendant la dernière itération, où elle en utilise sept. En regardant dans la table le nombre de cycles utilisés par l'instruction JR, nous pourrons vérifier l'existence de deux possibilités : si le branchement n'a pas lieu, JR requiert uniquement sept cycles. S'il a lieu, comme c'est généralement le cas, pendant la boucle, 12 cycles seront alors requis.

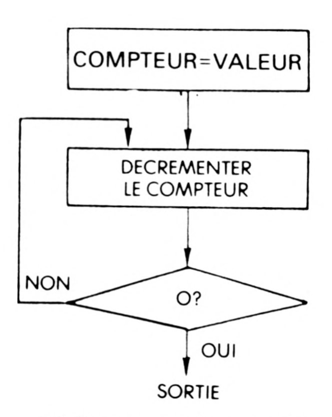


Figure 6.3. — Ordinogramme de base pour un délai

Le délai est donc de sept cycles pour la première instruction, plus 16 cycles pour les deux suivantes, multiplié par le nombre d'exécutions de la boucle, moins un délai de cinq cycles pour le dernier JR, puisqu'aucun branchement n'a lieu.

DELAI = 7 + 16 × 5 − 5 = 82 cycles

En supposant que les cycles sont de 0,5 microseconde, le délai programmé sera de 41 microsecondes.

Le type de boucle de délai que nous venons de décrire est utilisé dans la plupart des programmes d'entrée-sortie. Il est donc important d'en bien comprendre le mécanisme. Essayez d'effectuer les deux exercices suivants :

Exercice 6.1 : *Quels délais maximum et minimum peut-on obtenir avec ces trois instructions ?*

Exercice 6.2 : *Modifier le programme pour obtenir un délai d'environ 100 microsecondes.*

Pour obtenir un délai plus long, une solution simple consiste à introduire des instructions supplémentaires, entre le DEC et le J. Il est préférable d'introduire des instructions NOP (l'instruction NOP, de l'anglais No Opération, « ne rien faire », ne fait rien, en effet, et dure quatre cycles).

Obtention de délais plus longs

Il est possible de générer par programme des délais plus longs, en utilisant des compteurs plus grands. Une paire de registres permettra de contenir un compteur sur 16 bits. Pour simplifier, nous supposerons que la partie basse du compteur est « 0 ». L'octet haut est chargé avec la valeur « 255 » (le compte maximum), puis il est décrémenté. Chaque fois qu'il passe par 0, l'octet de poids fort est décrémenté de 1. Lorsque l'octet de poids fort atteint la valeur 0, et le programme s'arrête. Si une plus grande précision est requise dans la génération du délai, l'octet de poids faible pourra avoir une valeur non-nulle, au départ. Dans ce cas, nous pourrions écrire le programme exactement comme décrit ci-dessus, en ajoutant, derrière, le programme de trois lignes décrit ci-dessus.

Voici un programme de délai utilisant un compteur sur 24 bits :

```
DEL24      LD     B, CNT-HAUT   VALEUR DE L'OCTET DE
                                POIDS FORT (8 bits)
DEL16      LD     DE, − 1
A-BOUCLE   LD     HL, CNT-BAS   VALEUR DE LA PARTIE
                                BASSE (16 bits)
B-BOUCLE   ADD    HL, DE        LE DECREMENTER
           JR     C, B-BOUCLE   BOUCLER JUSQU'À CE
                                QU'IL SOIT NUL
           DJNZ   A-BOUCLE      DECREMENTE B ET
                                SAUTE
```

Remarquons que DE a été chargé avec la valeur − 1 pour que son addition à HL provoque la décrémentation de ce dernier.

Naturellement, des délais encore plus longs pourraient être générés en utilisant plus de trois mots. Le principe est le même que celui du compteur kilométrique sur une voiture. Lorsque la roue la plus à droite passe de « 9 » à « 0 », celle située immédiatement à sa gauche est incrémentée de 1. C'est le principe général du comptage utilisant plusieurs unités discrètes.

Le principal inconvénient de cette méthode est que le microprocesseur reste inoccupé pendant les centaines de millisecondes, voire même les secondes entières, où il s'emploie à générer des délais. S'il n'a effectivement rien à faire, c'est parfaitement acceptable. Mais, en général, le microprocesseur doit être disponible pour d'autres tâches, de sorte que les longs délais ne sont normalement pas générés par programme. En fait, même les délais brefs peuvent ne pas être acceptables, dans un système qui doit garantir des temps de réponse donnés dans certaines situations. Pour remédier à ce défaut, on recourt à des délais générés par hardware. De plus, la précision du délai risque d'être perdue si la boucle de comptage peut être interrompue. (Dans la mesure, bien sûr, où des interruptions sont mises en œuvre).

Exercice 6.3 : *Ecrire un programme qui génère un délai d'environ 100 millisecondes (délai caractéristique des échanges avec une télétype).*

Délais générés par hardware

Les délais sont obtenus, par hardware, au moyen de *générateurs programmables d'intervalles de temps :* « timers » en abrégé. [Nous utiliserons ici le mot anglais, car aucun équivalent français n'est couramment admis]. Un registre du timer est chargé avec une valeur. La différence avec la solution programmée est la suivante : le timer va automatiquement décrémenter le compteur, à un certain rythme. La période (entre deux décrémentations) peut généralement être choisie, ou ajustée, par le programmeur. Chaque fois que le timer atteint la valeur 0, il enverra normalement une interruption au microprocesseur. Il pourra aussi positionner un bit de statut, qui sera périodiquement testé par le microprocesseur. L'utilisation des interruptions sera expliquée plus avant dans ce chapitre.

D'autres modes d'opération peuvent être inclus dans le timer. Par exemple : démarrer de la valeur « 0 » et mesurer la durée d'un signal. Ou encore : compter le nombre d'impulsions reçues. Lorsqu'il fonctionne en générateur d'intervalles de temps, on dit que le timer opère en mode *réveil* (en anglais : one-shot, un coup). Lorsqu'il compte des impulsions, on dit qu'il travaille en mode *comptage d'impulsions.* Certains timers peuvent même contenir plusieurs registres, ainsi qu'un certain nombre de possibilités, sélectables par le programmeur.

Détection d'impulsions

Le problème de la détection d'impulsions est l'inverse de celui de la génération. Il comporte pourtant une difficulté supplémentaire : alors

qu'une impulsion est générée sous le contrôle du programme, les impulsions « en entrée » le sont par des dispositifs externes, de manière tout à fait *asynchrone* avec le déroulement du programme. Pour détecter une impulsion, deux méthodes peuvent être utilisées : *l'interrogation* (en anglais : polling), et les *interruptions.* Les interruptions seront présentées plus avant dans ce chapitre.

Voyons, pour l'instant, la méthode dite d'interrogation (polling). Elle consiste à lire, de manière continuelle, la valeur d'un registre d'entrée, en testant l'un de ses bits. Par exemple, le bit 0. Nous supposerons qu'à l'origine, le bit « 0 » est à la valeur 0. Lors de l'arrivée de l'impulsion, ce bit va prendre la valeur 1. La tâche du programme est de surveiller sans relâche le bit 0, jusqu'à ce qu'il prenne la valeur 1. Lorsque tel est le cas, l'impulsion a été détectée.

Voici le programme correspondant :

```
POLL    IN      A, (ENTREE)     LIRE LE REGISTRE D'EN-
                                TREE
        BIT     0, A            TESTER LE BIT 0
        JR      Z, POLL         BOUCLER TANT QU'IL EST
                                A 0
```

Inversement, supposons que la ligne d'entrée soit normalement à l'état 1, et que nous voulions détecter son passage à 0. C'est ce qui se passe, par exemple, avec la détection du bit de départ (START bit), lors du contrôle d'une ligne connectée à une télétype.

Le programme est le suivant :

```
POLL    IN      A, (ENTREE)     LIRE LE REGISTRE
                                D'ENTREE
        BIT     0, A            POSITIONNE L'INDICA-
        JR      NZ, POLL        TEUR
                                BOUCLER TANT QUE LE
                                BIT EST A 1
DEBUT   ...
```

Contrôle de la durée

Il est possible de contrôler la durée d'une impulsion reçue de la même manière que l'on calcule la durée d'une impulsion émise. Là encore deux techniques peuvent être utilisées : techniques programmée ou hardware. Lorsque la durée d'une impulsion est mesurée de façon programmée, un compteur est régulièrement incrémenté de 1, tant que la présence de l'impulsion est vérifiée. Tant qu'elle est présente, le programme boucle sur lui-même. Dès qu'elle disparaît, la valeur contenue dans le compteur est utilisée pour calculer la durée effective de l'impulsion.

```
DUREE    LD    B, 0            INITIALISE LE COMPTEUR
ATTENTE  IN    A, (ENTREE)     LIRE LE REGISTRE
         BIT   0, A
         JR    Z, ATTENTE      SAUT SI IMPULSION PAS
                               ARRIVEE
CALCUL   INC   B               INCREMENTER LE
                               COMPTEUR
         IN    A, (ENTREE)
         BIT   0, A            TESTER BIT 0
         JR    NZ, CALCUL      SAUT TANT QUE IMPUL-
                               SION PRESENTE
```

Naturellement, nous avons supposé ici que la durée maximum de l'impulsion n'est pas susceptible de provoquer un débordement du registre B. Si tel devait être le cas, il faudrait alors changer le programme, ou prendre en compte cette possibilité (sinon, il y aurait erreur de programmation !).

Nous savons, maintenant détecter et générer des impulsions, et allons nous efforcer de détecter et de générer de plus grandes quantités de données. Deux cas doivent être distingués : les échanges série et les échanges parallèles. Ensuite, nous appliquerons nos connaissances à l'utilisation de périphériques réels.

TRANSFERT PARALLELE D'UN MOT

Supposons que huit bits de données à transférer sont disponibles en parallèle (simultanément), à l'adresse « ENTREE » (voir figure 6.4). Le microprocesseur devra lire le mot de données à cette adresse, chaque fois qu'un mot d'état indiquera que des données sont disponibles (valides). Nous poserons que l'information d'état est contenue dans le bit 7 du mot d'adresse « ETAT ». Nous écrirons un programme qui lira et sauvera automatiquement les mots de donnée au fur et à mesure de leur arrivée. Pour simplifier, nous présumerons que le nombre de mots à lire est connu à l'avance, et contenu dans la case mémoire d'adresse « COMPTE ». Si cette information était inconnue, il faudrait alors tester chaque mot lu pour s'assurer qu'il est égal à un caractère, défini à l'avance, marquant la fin des données entrées. Ce caractère s'appelle *indicateur de fin de message* (en anglais « break character »). Il pourra s'agir de n'importe lequel des caractères possibles. Par exemple, le caractère d'effacement (rubout) ou, pourquoi pas, le caractère « * ». Nous savons déjà réaliser ce genre de test.

L'organigramme du programme est présenté à la figure 6.5. Il est évident. Nous allons tester l'information d'état jusqu'à ce qu'elle devienne « 1 », indiquant ainsi qu'un mot de donnée est prêt. Nous le lirons alors, et le rangerons en mémoire, à un emplacement prévu. Puis, nous décrémenterons le compteur, et testerons s'il passe à « 0 ». Si c'est le cas, l'opération est terminée ; sinon, il faudra attendre le mot suivant.

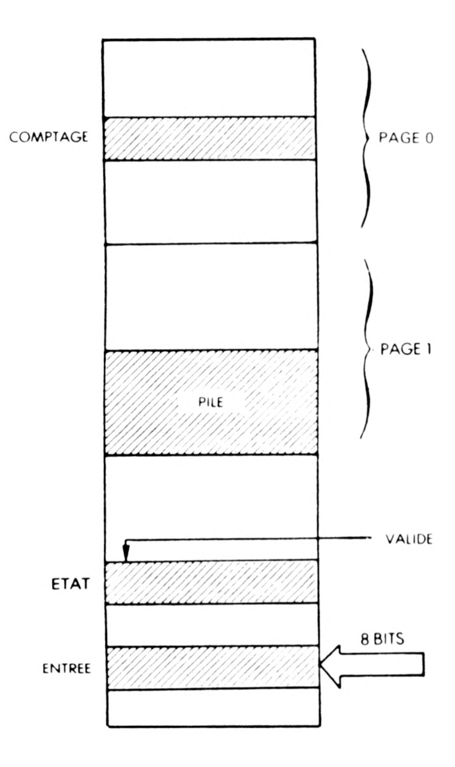


Figure 6.4. — Transfert parallèle d'un mot : la mémoire

Voici un programme simple fondé sur cet algorithme :

```
PARALLELE  LD    A, (COMPTE)    CHARGER A AVEC LE
                                COMPTE DE MOTS
           LD    B, A           B SERVIRA DE
                                COMPTEUR
ATTENTE    IN    A, (ETAT)      LIRE LE MOT D'ETAT
           BIT   7, A           VOIR SI UN MOT DE
                                DONNEE EST PRET
           JR    Z, ATTENTE     CONTINUER D'ATTEN-
                                DRE SI PAS PRET
           IN    A, (ENTREE)    LIRE LA DONNEE
           PUSH  AF             LA RANGER SUR LA
                                PILE
           DEC   B              DECREMENTER LE
                                COMPTEUR
           JR    NZ, ATTENTE    BOUCLER S'IL
                                N'ATTEINT PAS 0
```

Nous avons supposé que l'information « donnée prête » (data ready), reflétée par le bit 7 du mot d'état, est automatiquement remise à zéro après la lecture de ETAT. C'est généralement ce qui se passe dans les contrôleurs de périphériques.

SCRUTATION OU DEMANDE DE SERVICE

LIRE COMPTEUR
MOT PRET?
NON
OUI
TRANSFERER MOT
DECREMENTER COMPTEUR
NON
COMPTEUR = 0?
OUI
SORTIE

Figure 6.5. — Transfert parallèle de mots : ordinogramme

Les deux premières instructions initialisent le compteur B :

```
PARELLELE  LD    A, (COMPTE)
           LD    B, A
```

Remarquons qu'il n'existe pas de moyen simple de charger le registre B à partir du contenu d'une case mémoire. Il faut, soit commencer par charger A, avant de transférer son contenu dans B, soit charger simultanément les deux registres B et C.

Les trois instructions suivantes du programme lisent l'information d'état, et bouclent sur elles-mêmes tant que le bit 7 du mot d'état vaut « 0 ».

```
ATTENTE    IN    A, (ETAT)
           BIT   7, A           « IN » NE POSITIONNE
                                PAS LES INDICATEURS
           JR    Z, ATTENTE
```

Lorsque le saut n'a pas lieu, c'est qu'une donnée est valide, et qu'elle est susceptible d'être lue :

```
           IN    A, (ENTREE)
```

Le mot vient d'être lu, à l'adresse ENTREE, et il s'agit maintenant de le conserver. S'il s'avère que nous disposons de suffisamment de place sur la pile, nous pouvons utiliser :

```
PUSH          AF
```

qui rangera A (et F) sur la pile. Si la pile avait été pleine, ou le nombre de mots à transférer trop grand, nous n'aurions pas pu le mettre à cet endroit, et aurions dû nous résoudre à le transférer vers une zone donnée de la mémoire, en utilisant, par exemple, une instruction indexée. L'opération aurait nécessité une instruction supplémentaire pour incrémenter, ou décrémenter, le registre d'index. L'instruction PUSH est plus rapide (11 cycles d'horloge seulement).

Le mot de donnée a maintenant été lu et sauvé. Nous nous contenterons de décrémenter le compteur, et de tester si nous avons terminé :

```
              DEC    B
              JR     NZ, ATTENTE
```

La boucle continue tant que le compteur n'a pas atteint « 0 ».

Ce programme de neuf instructions pourrait être un *programme d'évaluation* (en anglais : benchmark). Un programme d'évaluation est soigneusement optimisé de façon à tester les possibilités d'un processeur donné, dans une situation donnée. [Exemple : les échanges en mode parallèle]. Il est conçu pour obtenir une vitesse et une efficacité maximales. Nous allons nous efforcer de calculer la vitesse de transfert maximum de ce programme, c'est-à-dire le nombre maximum de mots de données qui pourront être lus et rangés en mémoire en un temps donné. Nous supposerons que COMPTE se trouve en mémoire. La durée de chaque instruction peut être trouvée en consultant les tables situées au chapitre quatre.

Nous obtenons :

```
PARALLELE     LD     A, (COMPTE)          13
              LD     B, A                  4
ATTENTE       IN     A, (ETAT)            11
              BIT    7, A                  8
              JR     Z, ATTENTE         7/12
              IN     A, (ENTREE)          11
              PUSH   AF                   11
              DEC    B                     4
              JR     NZ, ATTENTE        7/12
```

Le temps d'exécution minimum est obtenu en supposant qu'une nouvelle donnée est prête chaque fois que le mot ETAT est testé. En d'autres termes, le premier JR rencontré ne doit jamais être effectué. La durée de lecture de tous les mots est alors :

$$13 + 4 + (11 + 8 + 7 + 11 + 11 + 4 + 12) \times \text{COMPTE}$$

En négligeant les dix-sept premiers cycles de l'initialisation du registre de comptage, le temps nécessaire au transfert un mot est de 64 cycles d'horloge, soit 32 microsecondes avec une horloge à 2 MHz.

Le rythme de transfert maximal est donc :

$$\frac{1}{32 \times (10^{-6})} = 31\text{K octets par seconde}$$

Exercice 6.4 : *Supposez que le nombre de mots à transférer soit supérieur à 256, et modifiez le programme en conséquence. Déterminez l'impact de cette modification sur le rythme maximal de transfert.*

Exercice 6.5 : *Modifiez le programme ci-dessus de sorte que sa vitesse soit améliorée :*

1. En utilisant JP au lieu de JR
2. En utilisant DJNZ
3. En utilisant INI ou IND

Le programme était-il vraiment optimal ?

Nous avons maintenant appris à transférer en parallèle des données à grande vitesse. Examinons maintenant un cas plus complexe.

TRANSFERT DE BITS EN SERIE

Une entrée série est une entrée où, contrairement à une entrée parallèle, les bits d'informations (0 ou 1) arrivent, successivement, sur la même ligne. Leurs arrivées respectives peuvent être séparées par des intervalles réguliers : c'est alors, normalement, une transmission *synchrone*. Elles peuvent avoir lieu par paquets, de manière irrégulière : c'est une transmission *asynchrone*. Nous allons développer un programme qui puisse marcher dans les deux cas. Le principe de récupération des données, lors d'une transmission série est simple : observer la ligne d'entrée (nous supposerons ici qu'il s'agit de la ligne 0). Lorsqu'un bit de donnée est détecté, sur cette ligne, il est lu, puis déposé dans un registre qui est, lui-même, décalé. Après lecture du huitième bit, l'octet ainsi constitué est rangé dans une case mémoire, de façon à être préservé. La constitution de l'octet suivant commence alors. Pour simplifier, nous supposerons que le nombre d'octets à recevoir est connu à l'avance. Nous serions autrement obligés, par exemple, de tester l'arrivée d'un caractère de fin de message pour stopper l'entrée des données. Ce que nous avons déjà appris à faire. L'organigramme du programme ci-dessous apparaît à la figure 6.6.

Voici le programme :

```
SERIE   LD    C, 0          INITIALISER LE MOT
                            D'ASSEMBLAGE
        LD    A, (COMPTE)
        LD    B, A          B EST LE COMPTEUR
                            D'OCTETS
BOUCLE  IN    A, (ENTREE)   LIRE LE PORT D'ADRESSE
                            ENTREE
        BIT   7, A          BIT 7 = BIT D'ETAT, BIT
                            0 = BIT DE DONNEE
        JR    Z, BOUCLE     ATTENDRE UN « 1 »
        SRL   A             METTRE LE BIT DE DON-
                            NEES DANS L'INDICA-
                            TEUR « CARRY »
        RL    C             ROTATION DU REGISTRE C
        JR    NC, BOUCLE    CONTINUER JUSQU'A CE
                            QUE 8 BITS SOIENT
                            ASSEMBLES
        PUSH  BC            SAUVER LE MOT SUR LA
                            PILE
        LD    C, 01         REINITIALISER LE BIT
                            INDICATEUR DE FIN
                            D'ASSEMBLAGE
        DEC   B             DECREMENTER LE
                            COMPTEUR
        JR    NZ, BOUCLE    ALLER ASSEMBLER LE
                            MOT SUIVANT SI PAS FINI
```

Ce programme a été conçu dans un souci d'efficacité. Il utilise de nouvelles techniques, que nous allons expliquer (voir figure 6.7).

Les conventions sont les suivantes : la case mémoire COMPTE est supposée contenir le nombre de mots à transférer. Le registre C sera utilisé pour assembler, en un octet, 8 bits consécutifs. L'adresse ENTREE représente un registre périphérique qui pourra être lu. Nous supposerons que le bit 7 est un bit d'état, ou bit d'horloge. Lorsqu'il vaut 0, la donnée n'est pas valide ; lorsqu'il vaut 1, la donnée est valide. Admettons que cette dernière arrive sur le bit 0 du registre ENTREE. Dans la réalité, le bit d'information et le bit de donnée proviennent rarement du même registre. Mais il est facile de modifier le programme en conséquence. Nous supposerons de plus que le premier bit de donnée est un « 1 ». Ce « 1 » indique que les données arrivent. Si tel n'était pas le cas, il faudrait, pour en tenir compte, mettre en œuvre une modification évidente. Nous y reviendrons. Le programme suit exactement le cheminement tracé par l'organigramme de la figure 6.6. Les premières lignes du programme forment une

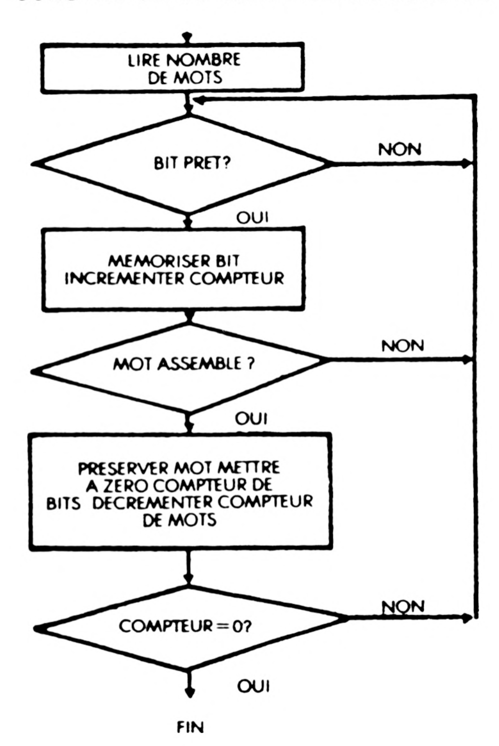


Figure 6.6. — Transfert de bits en série : ordinogramme

boucle qui attend qu'un bit de donnée soit prêt. Pour déterminer si un bit est prêt, il suffit de lire le contenu du registre ENTREE, puis de tester l'indicateur Z. Tant qu'il indique « 1 », le JR provoque un saut, et le programme se rebranche à l'adresse BOUCLE. Dès que le bit d'état [ou bit d'horloge] prend la valeur « 1 », alors le JR ne provoque plus de saut, et l'instruction suivante est exécutée.

Cette séquence initiale correspond à la flèche 1 de la figure 6.7.

A ce point, le bit 7 de l'accumulateur contient un 1, et le bit de donnée se trouve dans le bit 0. Le tout premier bit qui arrive doit être un « 1 ». Les suivants pourront être indifféremment des 0 ou des 1. Il s'agit maintenant de préserver le bit de donnée qui vient d'être lu.

L'instruction :

```
SRL A
```

décale le contenu de l'accumulateur d'une position vers la droite, en déposant le bit situé le plus à droite de A (celui qui nous intéresse : le bit 0)

dans l'indicateur de report (carry bit). Nous récupérerons, ensuite, le contenu de cet indicateur de report au moyen de l'instruction suivante (voir flèches 2 et 3 de la figure 6.7) :

RL C

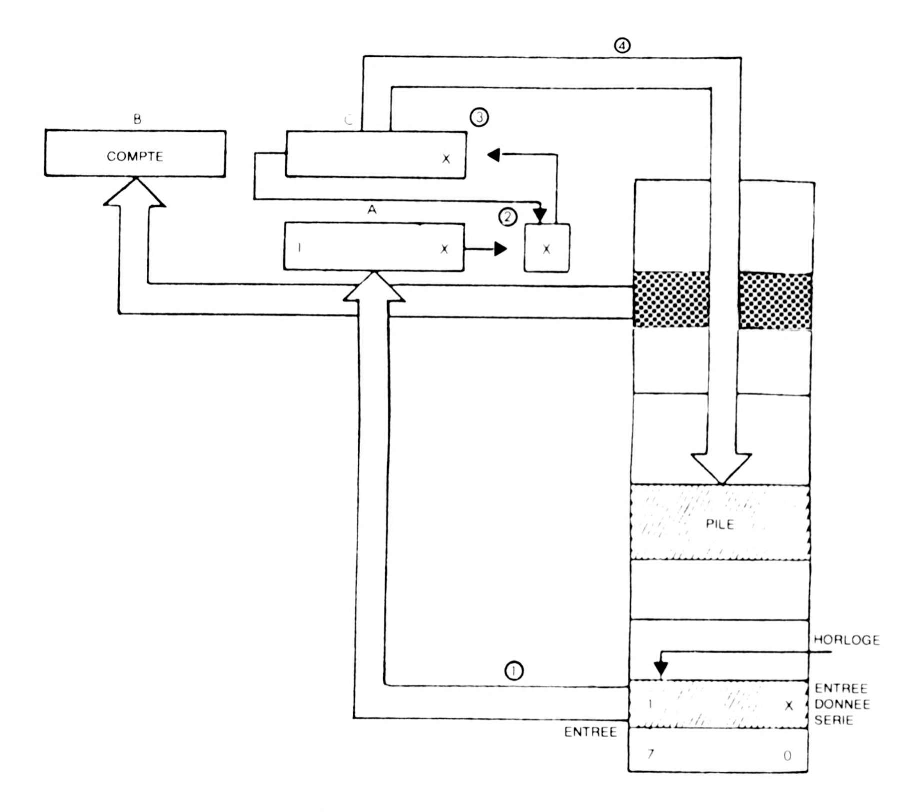


Figure 6.7. — Série parallèle : les registres

L'effet de cette instruction est de recopier le contenu de l'indicateur de report dans le bit de l'extrême-droite du registre C, et simultanément, de déposer le contenu du bit de l'extrême-gauche de C dans l'indicateur de report (si vous conservez quelques doutes sur le fonctionnement des instructions de rotation, reportez-vous au chapitre 4).

Précision importante : l'opération de rotation va, simultanément, sauver le contenu de l'indicateur de report (ici à l'extrême-droite de C).

Lors du premier passage à cette instruction, un « 0 » est déposé dans l'indicateur de report (à cause de la valeur initiale du registre C).

L'instruction suivante :

JR NC, BOUCLE

teste l'état de l'indicateur de report, et provoque un branchement à l'adresse BOUCLE, tant que cet indicateur vaut 0. Nous venons ici d'implanter un compteur automatique de bits. Il est aisé de se rendre compte qu'après le premier RL C, le registre C contiendra la valeur « 00000001 ». Huit décalages plus tard, le « 1 » « sortira » finalement du registre C, et « tombera » dans l'indicateur de report, empêchant ainsi que le branchement ait lieu. C'est une manière astucieuse de construire un compteur de boucle, en s'épargnant la peine de décrémenter le contenu du registre servant d'index. Cette technique permet de raccourcir le programme et d'améliorer sa vitesse.

Lorsque, finalement, l'instruction JR NC cesse de provoquer le saut, huit bits ont été assemblés dans le registre C. Cette valeur doit être préservée en mémoire, au moyen de l'instruction suivante (flèche 4 de la figure 6.7) :

```
PUSH BC
```

Le contenu de C (mais aussi de B) est ainsi sauvé, ce qui n'est possible que s'il reste suffisamment de place sur la pile. Si tel est bien le cas, nous recourrons à la manière la plus rapide de sauvegarder une information en mémoire, même si, par la même occasion, nous sauvons ainsi une information inutile (le contenu du registre B). En effet, le pointeur de pile est mis à jour automatiquement. Pour ranger le mot en mémoire sans utiliser la pile, il faudrait écrire une instruction supplémentaire pour mettre à jour notre pointeur. Il serait aussi, à la rigueur, possible d'utiliser l'adressage indexé, mais il faudrait alors incrémenter ou décrémenter le registre d'index. D'où une nouvelle perte de temps.

Maintenant que le premier mot de données est sauvé, il n'y a plus aucune garantie que le prochain bit de donnée qui sera lu soit un « 1 » plutôt qu'un « 0 ». Il est donc nécessaire de réinitialiser le registre C à la valeur « 00000001 », pour continuer à l'utiliser comme compteur. L'instruction suivante s'en chargera :

```
LD   C, 01
```

Pour finir, le compteur de mots va être décrémenté, puisqu'un mot complet vient d'être assemblé. Nous testerons ensuite s'il s'agit, ou non, de la fin du transfert :

```
DEC  B
JR   NZ, BOUCLE
```

Le programme précédent a été conçu pour travailler rapidement, et ainsi recevoir un flot rapide de bits de données. Lorsqu'il arrive à son terme, il faut, bien sûr, immédiatement recopier dans une zone mémoire appropriée ce qui vient d'être déposé sur la pile. Nous avons vu au chapitre 5 la manière de procéder pour réaliser un tel transfert de blocs.

Exercice 6.6 : *Calculez la vitesse maximale à laquelle ce programme pourra lire des bits de données arrivant en série. On s'informera, dans la table à la fin du livre, du nombre de cycles d'horloge requis par chaque instruction, puis on calculera le temps nécessaire à l'exécution du programme. Pour calculer la durée d'une boucle, on multipliera simplement le temps d'exécution de la boucle par le nombre d'exécutions. Le temps d'exécution, exprimé en microsecondes, sera égal au nombre de cycles d'horloge, multiplié par la durée en microsecondes d'un cycle. Pour obtenir la vitesse maximale d'exécution, on supposera que, chaque fois que le registre d'entrée est testé, un nouveau bit est prêt.*

Ce programme est plus difficile à comprendre que les précédents. Nous allons le réexaminer en détail, et en expliquer quelques mécanismes.

Un bit de données arrive, de temps en temps, sur le bit 0 du registre ENTREE. Nous pourrions avoir, par exemple, trois « 1 » successifs. Il nous faut donc un *moyen de les distinguer.* C'est la fonction du signal « d'horloge ».

Le signal d'horloge (ou d'état) signale l'arrivée de chaque nouveau bit. Avant de lire un bit, il est donc nécessaire de tester le bit d'état. S'il vaut « 0 », il faut attendre. S'il vaut « 1 », un nouveau bit doit être lu.

Nous avons supposé ici que le bit d'état est connecté au bit 7 du registre ENTREE.

Exercice 6.7 : *Dites pourquoi le bit 7 est utilisé pour l'information d'état, et le bit 0 pour les données. Cela a-t-il de l'importance ?*

Après la lecture d'un bit de données, il est indispensable de le préserver dans un octet, puis de décaler ce dernier à gauche pour y mettre le prochain bit.

Malheureusement, l'accumulateur est déjà utilisé pour lire et tester les données et l'état du registre périphérique. Si nous assemblions les bits de donnée dans l'accumulateur, le bit 7 serait effacé par la prochaine lecture du bit d'état.

Exercice 6.8 : *Suggérez un moyen de tester l'état du registre sans effacer le contenu de l'accumulateur (une instruction spéciale ?). Dans l'hypothèse favorable, est-il possible d'utiliser l'accumulateur pour assembler les bits successifs, à leur arrivée ? Pouvez-vous améliorer la vitesse en utilisant une instruction « de saut automatique » ?*

Exercice 6.9 : *Réécrivez le programme, en utilisant l'accumulateur pour assembler les bits. Comparez-le au précédent, en termes de vitesse et de nombre d'instructions.*

Nous allons apporter deux modifications au programme précédent.

Dans notre exemple particulier, le tout premier bit qui arrive est un signal spécial dont il est avéré qu'il est un « 1 ». En général, il pourrait être n'importe quoi.

Exercice 6.10 : *Modifiez le programme ci-dessus, en supposant que le tout premier bit qui arrive est une donnée valide à conserver, dont la valeur est soit « 1 », soit « 0 ». Indication : le compteur de bits devrait encore fonctionner correctement, s'il est initialisé à la bonne valeur.*

Les octets assemblés sur la pile ont été sauvés, pour gagner du temps. Nous aurions naturellement pu les ranger en mémoire.

Exercice 6.11 : *Modifiez le programme ci-dessus de manière à ranger en mémoire, à partir de l'adresse BASE, les mots assemblés.*

Exercice 6.12 : *Modifiez le programme ci-dessus pour que le transfert s'arrête lors de la rencontre du caractère S.*

La solution hardware

De même que pour la plupart des algorithmes d'entrée/sortie, il est possible de fabriquer des circuits capables d'exécuter celui-ci (solution hardware). Un tel circuit est appellé UART (de l'anglais Universal Asynchronous Receiver-Transmitter, qui signifie émetteur-récepteur asynchrone universel). Pour maintenir au minimum le prix des composants, il conviendra d'utiliser le programme ci-dessus, ou l'un de ses dérivés.

Exercice 6.13 : *Modifiez le programme en supposant que les données proviennent du bit 0 du registre périphérique d'adresse ENTREE, et l'information d'état du bit 0 de l'adresse ENTREE + 1.*

CONCLUSION SUR LES OPÉRATIONS ÉLÉMENTAIRES D'E/S

Nous avons maintenant appris à programmer les opérations élémentaires d'entrées/sorties, et à gérer des flots de données, en parallèle et en série. Nous allons maintenant nous intéresser à l'utilisation des organes d'entrée-sortie réels.

COMMUNIQUER AVEC DES ORGANES D'ENTRÉE/SORTIE

Pour échanger des données avec des organes d'entrées/sorties, il importe d'abord de s'assurer de la présence de données valides dans l'hypothèse où elles doivent être lues, et également de s'assurer que l'organe de sortie est prêt à recevoir des données, s'il est nécessaire d'en envoyer. Deux procédés possibles : les protocoles et les interruptions.

Les protocoles

Les protocoles permettent, en général, d'établir une communication entre deux systèmes asynchrones [deux systèmes qui ne sont pas synchronisés]. Par exemple, pour envoyer un mot à une imprimante.

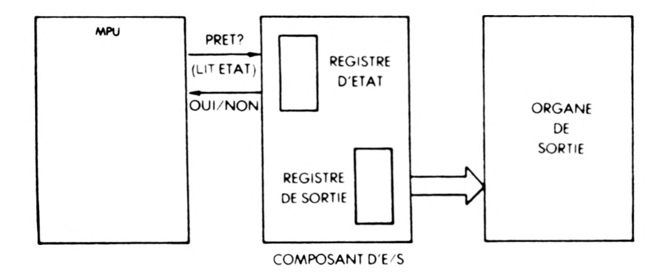


Figure 6.8. — Protocoles d'échange (Sortie)

Travaillant en mode parallèle, il faut d'abord s'assurer que sa mémoire-tampon d'entrée (input buffer) est disponible. Nous demanderons donc à l'imprimante : « es-tu prête ? » L'imprimante répondra « oui » ou « non ». Si elle répond non, il faudra attendre. Si elle répond oui, nous pourrons envoyer les données (voir figure 6.8).

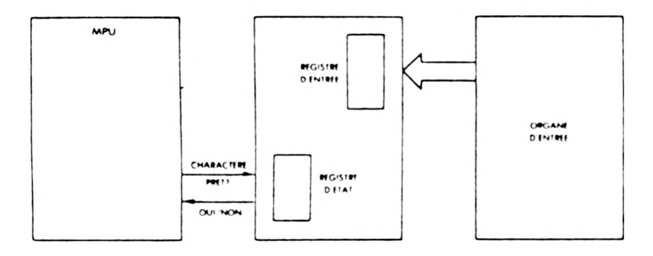


Figure 6.8a. — Protocoles d'E/S : Entrée

Inversement, avant de lire les données provenant d'un périphérique d'entrée, il faut vérifier s'il s'y trouve, des données valides. Nous demanderons : « y a-t-il des données valides ? », et le périphérique nous répondra « oui » ou « non ». Ces « oui » et ces « non » peuvent être matérialisés par des bits d'état, ou par d'autres moyens (voir figure 6.8a).

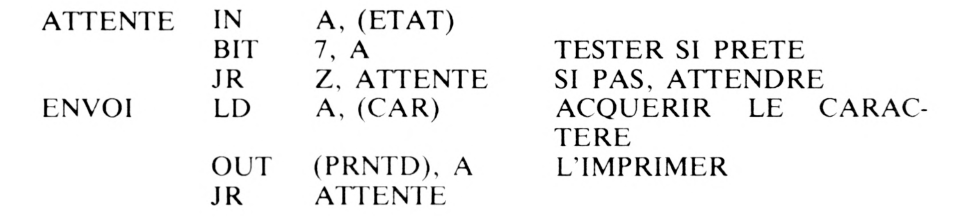
Par analogie, pour échanger des informations avec une personne indépendante, qui peut fort bien être occupée à tout autre chose à ce moment, vous devez vous assurer qu'elle est prête à discuter avec vous. Les règles de courtoisie élémentaires veulent que vous lui posiez la question de savoir si elle est disponible. Ensuite, vous pourrez commencer à discuter. La même procédure est utilisée pour les échanges de données avec les périphériques d'entrées/sorties.

Illustrons cette procédure, à l'aide d'un exemple simple.

Envoi d'un caractère à l'imprimante

Nous supposerons que le caractère est rangé en mémoire à l'adresse CAR.

Le programme d'impression est le suivant :



```
ATTENTE  IN    A, (ETAT)
         BIT   7, A          TESTER SI PRETE
         JR    Z, ATTENTE    SI PAS, ATTENDRE
ENVOI    LD    A, (CAR)      ACQUERIR LE CARAC-
                             TERE
         OUT   (PRNTD), A    L'IMPRIMER
         JR    ATTENTE
```

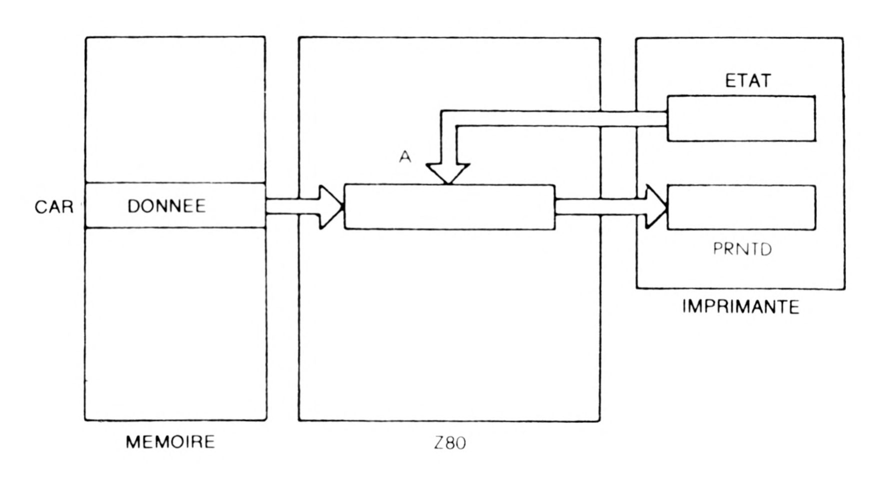


Figure 6.9. — Imprimante : les chemins de données

Le programme d'impression est évident. Il utilise le protocole déjà décrit. Sur la figure 6.9 sont figurés les chemins de données.

Le caractère, appelé DONNÉE, est rangé à l'adresse mémoire CAR. Il faut d'abord tester l'état de l'imprimante. Dès que le bit 7 du registre d'état passe à 1, cela signifie que l'imprimante est prête pour une sortie. Autrement dit que son buffer de sortie est disponible. A ce point, le caractère est chargé dans l'accumulateur, puis envoyé à l'imprimante, via l'accumulateur. Aussi longtemps que le bit d'état reste à 0, le programme boucle à l'adresse ATTENTE.

Exercice 6.14 : *Combien d'instructions seraient-elles économisées, dans le programme ci-dessus, s'il était possible de charger les données directement dans le registre C, et d'envoyez le contenu de C vers l'imprimante ?*

Exercice 6.15 : *Pour utiliser une imprimante, il est généralement nécessaire de lui envoyer une commande de démarrage préalable. Vous modifierez le programme de façon qu'il génère une telle commande, en supposant que l'envoi de la commande est opéré en écrivant un 1 dans le bit 0 du registre ETAT. Ce dernier sera supposé bidirectionnel.*

Exercice 6.16 : *Si l'instruction BIT n'existait pas sur le Z80, pourriez-vous utiliser à la place, à la ligne 2 du programme, une autre instruction ? Si oui, expliquez l'avantage éventuel de l'instruction BIT.*

Exercice 6.17 : *Modifiez le programme ci-dessus pour écrire une chaîne de n caractères, n étant supposé inférieur à 255.*

Exercice 6.18 : *Modifiez le programme ci-dessus pour imprimer une chaîne de caractères se terminant par le caractère « retour-chariot » (carriage return).*

Nous allons maintenant compliquer la procédure de sortie, en utilisant une conversion de code, et en envoyant plusieurs lignes à la fois.

Sortie vers une LED 7 segments

Une diode électro-luminescente (LED) traditionnelle à sept segments permet d'afficher les chiffres de « 0 » à « 9 », et même les chiffres hexadécimaux de « 0 » à « F », en combinant les segments éclairés. Une LED 7 segments est montrée sur la figure 6.10. Les caractères susceptibles d'être générés par elle sont présentés à la figure 6.11.

Les segments de la LED sont numérotés de A à G (voir figure 6.10).

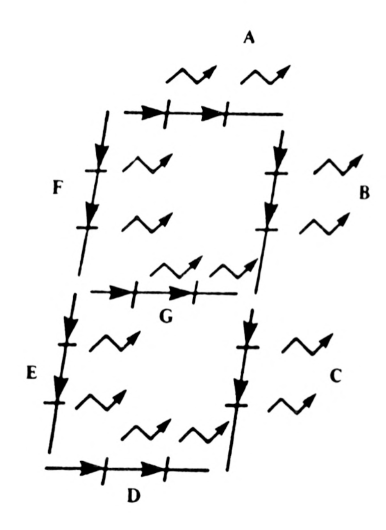


Figure 6.10. — Une LED 7 segments

Par exemple, « 0 » sera affiché en allumant les segments abcdef. Supposons maintenant que le bit 0 d'un port de sortie soit connecté au segment a, le bit 1 au segment b, et ainsi de suite. Le bit 7 n'est pas utilisé. Le code binaire requis pour allumer les segments fedcba (pour afficher « 0 ») est donc « 0111 1111 ».

En hexadécimal : 3FH. Voyons l'exercice suivant.

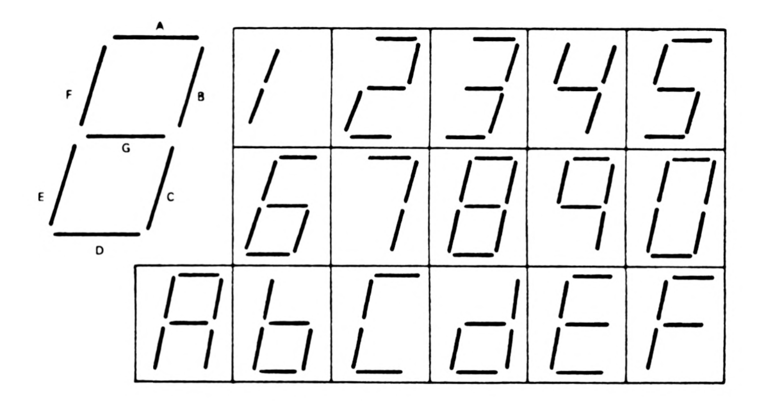


Figure 6.11. — Caractères hexadécimaux générés avec une DEL 7 segments

Exercice 6.19 : *Calculez les codes permettant d'afficher, sur la LED, les chiffres hexadécimaux de « 0 » à « F ». Remplissez la table ci-dessous :*

Hex	Code LED	Hex	Code LED	Hex	Code LED	Hex	Code LED
0	3F	4		8		C	
1		5		9		D	
2		6		A		E	
3		7		B		F	

Nous allons maintenant afficher des valeurs hexadécimales sur plusieurs LED.

Gestion de plusieurs LED

Une LED n'a pas de mémoire. Elle n'affiche les données qu'aussi longtemps que ses segments sont activés. Pour conserver un coût bas à l'affichage à base de LED, le microprocesseur affichera les données *tour à tour sur chacune des LED.*

La rotation entre les LED doit être suffisamment rapide pour éviter le scintillement apparent. Ce qui implique que le temps de passage d'une LED à la suivante soit inférieur à 100 millisecondes. Nous allons écrire un programme capable de respecter cette condition. Le registre C contiendra l'adresse de la LED sur laquelle un chiffre doit être écrit. L'accumulateur contiendra la valeur hexadécimale à envoyer. Notre première tâche sera de convertir la valeur hexadécimale en représentation 7 segments. Nous avons, au paragraphe précédent, construit la table d'équivalence. Puisqu'il nous faut accéder à cette table, nous utiliserons l'adressage indexé. Le déplacement de l'index nous sera fourni par la valeur hexadécimale. Cela signifie que le code de la représentation 7 segments du chiffre « 3 » se trouve dans le troisième élément de la table, après l'adresse de base. Nous appellerons SEGBAS cette adresse de base.

Le programme est le suivant :

```
DEL     LD    E, A            A CONTIENT LE CHIFFRE
                              HEXADECIMAL
        LD    D, 0            DE CONTIENT LE DEPLA-
                              CEMENT
        LD    HL, SEGBAS      UTILISER HL COMME
                              INDEX
        ADD   HL, DE          HL CONTIENT L'ADRESSE
                              DU CODE
        LD    A, (HL)         LIRE LE CODE DANS LA
                              TABLE
        LD    B, 50H          VALEUR DE DELAI
                              (N'IMPORTE QUEL NOM-
                              BRE ASSEZ GRAND)
DELAI   OUT   (C), A          SORTIR LE CARACTERE
        DEC   B               DECREMENTER LE
                              COMPTEUR DE DELAI
        JR    NZ, DELAI       ET ATTENDRE LA FIN
        DEC   C               C EST LE NUMERO DU
                              PORT
        LD    A, C
        CP    MINDEL          VIENT-ON D'ECRIRE SUR
                              LA DERNIERE DEL
        JR    NZ, SORTIE
        LD    BC, (MAXDEL)    SI OUI, REMETTRE C SUR
                              LA PREMIERE
SORTIE  RET
```

Le programme utilise le registre C pour contenir l'adresse de la DEL à allumer, et l'accumulateur A pour contenir le chiffre à afficher.

Le programme cherche d'abord le code correspondant à la valeur hexadécimale contenue dans l'accumulateur. Le registre DE est utilisé

comme valeur de déplacement, et le registre HL comme index 16 bits. Le code du chiffre hexadécimal est additionné à l'adresse de base de la table :

```
DEL     LD    E, A          CODE 7-SEGMENTS
        LD    D, 0
        LD    HL, SEGBAS
        ADD   HL, DE
```

Une boucle est ensuite initialisée de telle manière que le code obtenu dans la table soit affiché pendant une durée correcte. Nous avons ici choisi, arbitrairement, la constante 50 hexadécimale :

```
        LD    A, (HL)       LIRE LE CODE DANS LA
                            TABLE
        LD    B, 50 H       VALEUR DE DELAI
```

Le délai est obtenu avec une boucle classique. La première instruction :

```
DELAI   OUT   (C), A
```

envoie le contenu de l'accumulateur vers le port d'E/S pointé par le registre C (le numéro de la LED). Les deux instructions suivantes forment la boucle de délai :

```
        DEC   B
        JR    NZ, DELAI
```

Une fois le délai terminé, il convient de pointer sur une autre DEL, par exemple en décrémentant le numéro de la DEL et en s'assurant bien sûr qu'on ne dépasse pas le plus petit numéro :

```
        DEC   C
        LD    A, C
        CP    MINDEL
        JR    NZ, SORTIE
        LD    BC, (MAXDEL)
SORTIE  RET
```

Nous avons ici supposé que le programme avait été écrit sous forme d'un sous-programme, et la dernière instruction est donc RET (« retour de sous-programme »).

Exercice 6.20 : *Il est généralement nécessaire d'éteindre tous les segments avant d'afficher un chiffre. Modifier le programme précédent pour le faire (on enverra d'abord le code caractère 00 avant d'afficher le chiffre désiré).*

Exercice 6.21 : *Qu'arriverait-il à l'afficheur 7 segments si l'étiquette DELAI était descendue d'une instruction ? Cela changerait-il la durée des événements ? Cela changerait-il l'apparence de l'afficheur ?*

Exercice 6.22 : *Vous remarquerez qu'en fait les quatre premières instructions du programme réalisent un adressage indexé sur 16 bits de la mémoire. Mais, chose étrange, sans utiliser le dispositif normal d'indexation. Supposons que l'adresse SEGBAS est connue d'avance. Appelez SEGBSH la partie haute de cette adresse et SEGBSL la partie basse. Ranger SEGBSH dans la partie haute du registre d'index IX. Maintenant, réécrivez le programme en utilisant SEGBSL comme valeur de déplacement. Quels sont les avantages et les inconvénients de cette approche ?*

Exercice 6.23 : *Si, par hypothèse, le programme ci-dessus est un sous-programme, vous remarquerez qu'il utilise les registres B, D, E, H et L et en modifie les contenus. En supposant que le sous-programme dispose librement de cinq adresses mémoires appelées T1, T2, T3, T4 et T5, ajoutez, au début et à la fin du programme, des instructions qui garantissent qu'au retour du sous-programme, les contenus des registres B, D, E, H et L seront les mêmes qu'à l'entrée.*

Exercice 6.24 : *Même exercice que ci-dessus, mais en supposant que le sous-programme ne dispose pas des cases mémoire T1 à T5. (Indication : souvenez-vous qu'il existe dans tout ordinateur un mécanisme permettant de sauver de l'information de manière chronologique).*

Nous avons maintenant résolu des problèmes courants d'entrées-sorties. Envisageons le cas d'un périphérique commun : la télétype.

Entrées-sorties sur télétype

La télétype est un périphérique série. Elle émet et reçoit des mots d'information, sous forme de suites de bits. Chaque caractère est codé sur 8 bits, selon le code ASCII (voir la table des codes ASCII à la fin du livre). Il est, de plus, précédé d'un « bit de début » (start bit), et suivi de deux « bits de fin » (stop bits). Selon l'interface la plus utilisée, appelée boucle de courant 20 milliampères, l'état de la ligne est normalement à 1. Le processeur peut ainsi vérifier que la ligne n'a pas été coupée. Un bit de début est une transition de 1 à 0, qui indique au récepteur que des bits de donnée vont suivre. La vitesse de la télétype standard est de 10 caractères par seconde. Nous venons de voir que chaque caractère échangé est composé de 11 bits : la télétype peut donc transmettre 110 bits par seconde. Nous appellerons baud l'unité 1 bit par seconde, et nous dirons que la télétype fonctionne à 110 bauds. A présent, nous allons nous efforcer de développer un programme capable de sérialiser des bits pour la télétype à la vitesse correcte.

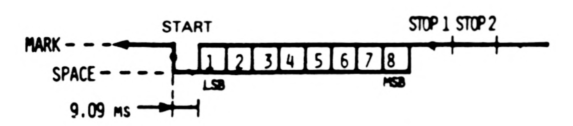


Figure 6.12. — Format d'un mot sur télétype

Une vitesse de 110 bits par seconde implique que les bits s'échelonnent tous les 9,09 millisecondes. Ce sera la durée de la boucle de délai à réaliser entre les bits successifs. Le format d'un mot, pour la télétype, est présenté à la figure 6.12. L'ordinogramme d'entrée des bits depuis la télétype est à la figure 6.13. Le programme est le suivant :

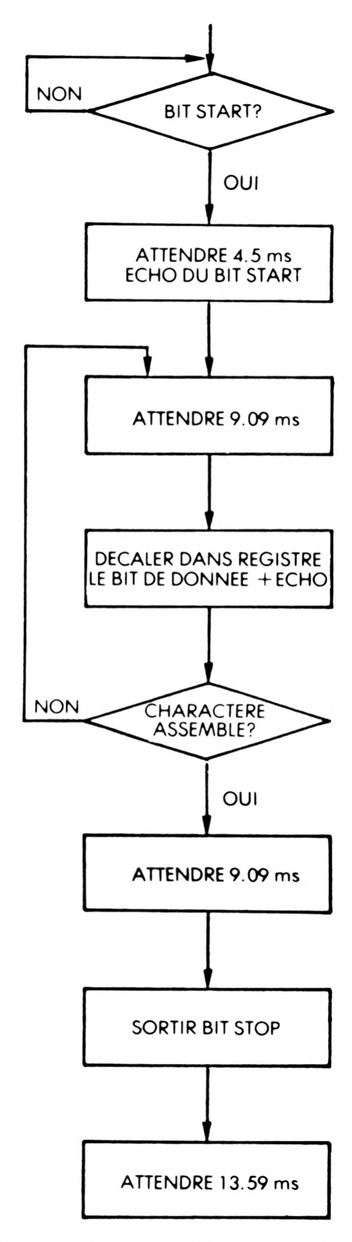


Figure 6.13. — Entrée télétype avec écho

```
TTYIN    IN    A, (ETAT)
         BIT   7, A          DONNEE PRETE ?
         JR    Z, TTYIN      SINON ATTENDRE
         CALL  DELAI1        ATTENDRE LE CENTRE
                             DE L'IMPULSION
         IN    A, (TTYBIT)   « START BIT »
         OUT   (TTYBIT), A   FAIRE L'ECHO
         CALL  DELAI9        IMPULSION SUIVANTE
                             (9MS)
         LD    B, 08H        COMPTE DE BITS
SUIVANT  IN    A, (TTYBIT)   LIRE LE BIT DE DONNEE
         OUT   (TTYBIT), A   EN FAIRE L'ECHO
         SRL   A             LE SAUVE DANS LE
                             « CARRY »
         RR    C             PUIS DANS LE REGIS-
                             TRE C
         CALL  DELAI9        ATTENDRE IMPULSION
                             SUIVANTE
         DEC   B             DECREMENTER COMPTE
                             DE BITS
         JR    NZ, NEXT
         IN    A, (TTYBIT)   LIRE LE « STOP BIT »
         OUT   (TTYBIT), A   EN FAIRE L'ECHO
         CALL  DELAI9        SAUTER LE SECOND STOP
                             BIT
         RET
```

Figure 6.14. — Programme de lecture télétype

Examinons en détail ce programme. Il convient, d'abord, de tester l'état de la télétype, pour établir si un caractère est disponible :

```
TTYIN    IN    A, (ETAT)
         BIT   7, A
         JR    Z, TTYIN
```

L'instruction BIT est une possibilité intéressante du Z80, qui permet de tester n'importe quel bit dans n'importe quel registre. Elle ne modifie pas le contenu du registre testé. L'indicateur Z est mis à 1 si le bit testé est à 0, et réciproquement.

Le programme va ainsi boucler jusqu'à ce que, finalement, le bit d'état passe à 1. Il s'agit d'une boucle d'interrogation classique.

A noter que, dans la mesure où l'information d'état n'a pas à être préservée, nous aurions pu avantageusement utiliser :

```
         AND   10000000B
```

au lieu de :

```
         BIT   7, A
```

Nous aurions économisé 4 cycles T. Mais cette instruction détruit le contenu de l'accumulateur (ce qui est acceptable ici). En optimisant un programme, il faut garder à l'esprit que toute nouvelle instruction peut introduire des effets de bords. Nous avons ensuite une boucle d'attente de 4,5 ms. Le bit de donnée ne sera lu qu'au milieu de l'impulsion.

```
        CALL  DELAI1
```

DELAI1 est le sous-programme qui réalise le délai recherché. Le premier bit qui arrive est le bit de début (start bit). Il ne nous sert à rien, mais il doit, au moins, être renvoyé à la télétype, au moyen des deux instructions suivantes :

```
TTYIN   IN    A, (TTYBIT)
        OUT   (TTYBIT), A
```

Il faut ensuite attendre le premier bit. Le délai nécessaire, 9,09 millisecondes, est réalisé par le sous-programme DELAI9 :

```
        CALL  DELAI9
```

Le registre B est utilisé comme compteur, et chargé avec la valeur 8, de façon à compter la lecture de 8 bits.

```
        LD    B, 08H
```

Chaque bit de donnée est lu, à son tour, dans l'accumulateur, puis renvoyé à la télétype. Nous supposerons que les bits de donnée arrivent sur le bit 0 de l'accumulateur. Le bit de donnée est ensuite préservé dans le registre C. Il y sera entré par l'entremise d'une rotation vers la droite de 1 position du contenu du registre C. Le transfert du bit de donnée de A vers C se fait via l'indicateur de report (carry).

```
SUIVANT IN    A, (TTYBIT)
        OUT   (TTYBIT), A
        SRL   A
        RR    C
```

Cette séquence est illustrée à la figure 6.15.

Le délai habituel de 9 millisecondes est réalisé, et le compteur de bits décrémenté. La boucle continue jusqu'à ce que 8 bits aient été lus.

```
        CALL  DELAI9
        DEC   B
        JR    NZ, SUIVANT
```

Pour finir, le bit de fin (stop bit) est lu, et l'écho en est fait. Il est généralement suffisant de ne renvoyer qu'un stop bit ; il aurait été possible,

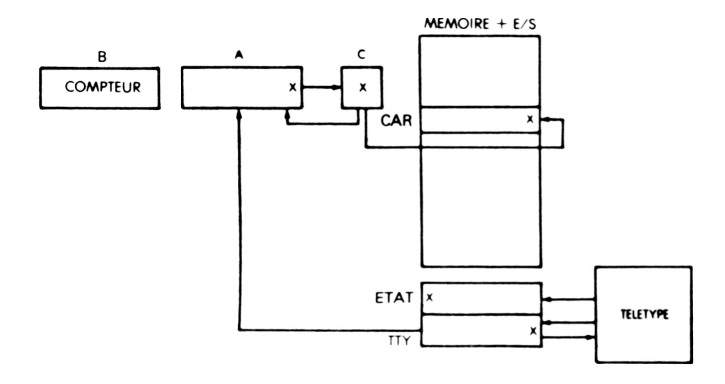


Figure 6.15. — Entrée télétype

cependant, au prix de deux instructions supplémentaires, de renvoyer les deux.

```
IN      A, (TTYBIT)
OUT     (TTYBIT), A
CALL    DELAI9
RET
```

Il faut examiner ce programme avec attention. La logique en est simple. Ce qui, par contre, est nouveau, c'est que chaque bit lu depuis la télétype (à

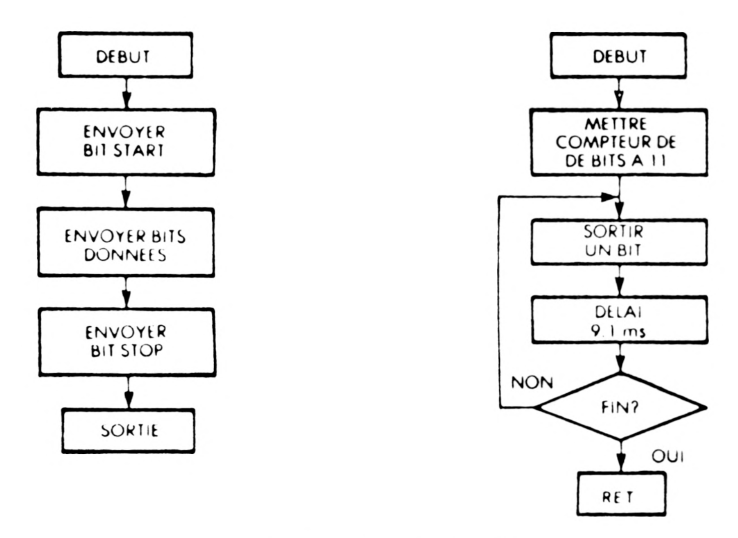


Figure 6.16. — Sorties télétype

l'adresse TTYBIT) doit être renvoyé à cette dernière. Il s'agit là d'une caractéristique standard de la télétype. Lorsque l'utilisateur appuie sur une touche, l'information est transmise au processeur, qui la renvoie au mécanisme d'impression de la télétype, afin de vérifier que les lignes de transmission fonctionnent bien, et que le processeur travaille. En effet, puisque les touches du clavier ne commandent pas le mécanisme d'impression, il faut nécessairement que l'ensemble lignes + processeur fonctionne pour que s'imprime sur le papier le caractère qui vient d'être frappé sur la télétype.

Exercice 6.25 : *Ecrire un sous-programme qui produise un délai de 9,09 millisecondes.*
(le sous-programme DELAI9).

Exercice 6.26 : *En utilisant l'exemple du programme ci-dessus, écrire un programme PRINTC, qui envoie sur la télétype le contenu de la case mémoire d'adresse CHAR.*

Voici la réponse :

```
PRINTC   LD     B, 11           COMPTEUR = 11 BITS
         LD     A, (CHAR)       ALLER CHERCHER LE
                                CARACTERE
         OR     A               METTRE L'INDICATEUR
                                « CARRY » A 0
         RLA                    ENTRER L'INDICATEUR
                                « CARRY » DANS A
SUIVANT  OUT    (TTYBIT) A      ENVOI DU BIT 0
         CALL   DELAI           9 MILLISECONDES
         RRA                    PRESENTER LE BIT
                                SUIVANT
         SCF                    CARRY = 1 (POUR LES
                                STOP BITS)
         DEC    B               DECREMENTER
                                COMPTEUR DE BITS
         JR     NZ, SUIVANT
```

Le registre B sert de compteur de bits pour la transmission. Le contenu du bit 0 de A est envoyé sur la ligne de la télétype (« TTYBIT »). A noter que l'indicateur de report (carry) est mis à 0 par l'instruction :

```
OR      A
```

A la fin du programme, le bit de report [carry] est mis à 1 par :

```
SCF
```

afin de produire les stop bits.

Exercice 6.27 : *Modifiez le programme de telle façon qu'il attende un bit de début (start bit) à la place du bit d'état.*

Ecrire une chaîne de caractères

Nous supposerons que le programme PRINTC (voir exercice 6.26) s'occupe d'envoyer un caractère à la télétype, ou d'ailleurs à n'importe quel périphérique. Nous écrirons un programme qui envoie les contenus des adresses mémoires DÉBUT à DÉBUT + N.

Le programme est évident (voir figure 6.17) :

```
PCHAINE LD    B, NBR          LONGUEUR DE LA
                              CHAINE
        LD    HL, DEBUT       ADRESSE DE BASE
SUIVANT LD    A, (HL)         CHARGER LE CARAC-
                              TERE
        CALL  PRINTC          L'IMPRIMER
        INC   HL              POINTER SUR LE CARAC-
                              TERE SUIVANT
        DEC   B
        JR    NZ, SUIVANT     ET BOUCLER
        RET
```

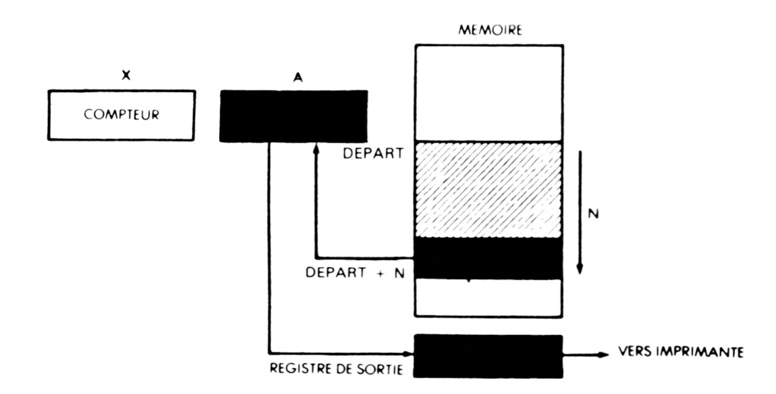


Figure 6.17. — Impression d'un bloc mémoire

CONCLUSION SUR LES PÉRIPHÉRIQUES

Nous avons décrit les techniques de base de la programmation des échanges avec les périphériques classiques. En plus du transfert des données, il est nécessaire de gérer, pour chaque périphérique, un ou plusieurs registres de contrôle, afin de définir la vitesse de l'échange, le

mécanisme d'arrêt et diverses autres options. Avant de programmer les échanges avec un périphérique, il est conseillé de consulter la notice d'utilisation. (Pour plus de détails sur les algorithmes spécifiques de communication avec les périphériques usuels, le lecteur se reportera à notre livre C5 : *Techniques d'Interface des Microprocesseurs*).
Nous savons donc, désormais, gérer des périphériques isolés. En fait, dans un vrai système, tous les périphériques sont connectés aux bus, et peuvent exiger d'être servis en même temps. Comment, dès lors, répartir l'emploi du temps du microprocesseur ?

GESTION DES ENTRÉES/SORTIES

Plusieurs demandes d'entrées-sorties peuvent être émises simultanément. Un mécanisme doit donc être implanté dans chaque système pour fixer l'ordre dans lequel les demandes devront être satisfaites. Trois techniques de base sont employées dans ce but. Elles peuvent être combinées entre elles. Ce sont : l'interrogation, les interruptions, et l'accès direct mémoire (ADM, en anglais DMA direct memory access). Nous ne décrirons que les deux premières. L'ADM, technique purement hardware, ne retiendra pas, ici, notre attention (elle est présentée dans nos livres).

L'INTERROGATION

Conceptuellement, l'interrogation est la méthode de gestion de plusieurs périphériques la plus simple. Dans cette stratégie, le processeur interroge successivement les périphériques connectés à ses bus. Si un périphérique demande un service, il le sert. S'il ne demande rien, le processeur examine le suivant. La méthode d'interrogation n'est pas utilisée pour les seuls périphériques, mais aussi pour *toutes les routines de service de périphériques.*

Par exemple, si le système est équipé d'une télétype, d'un enregistreur de bandes et d'un écran type CRT, le programme d'interrogation questionnera d'abord la télétype : « As-tu un caractère à transmettre ? », puis la routine de sortie vers la télétype « as-tu un caractère à envoyer ? », puis, en supposant que les réponses aient jusqu'ici été négatives, les routines de l'enregistreur de bandes, et finalement de l'écran CRT. Cette méthode peut être employée, même avec un seul périphérique connecté au système.

A titre d'exemple, vous trouverez aux figures 6.20 et 6.21, les ordinogrammes de lecture d'un ruban papier et d'écriture sur une imprimante.

Exemple : une boucle d'interrogation pour les périphériques 1,2,3,4 (voir figure 6.19)

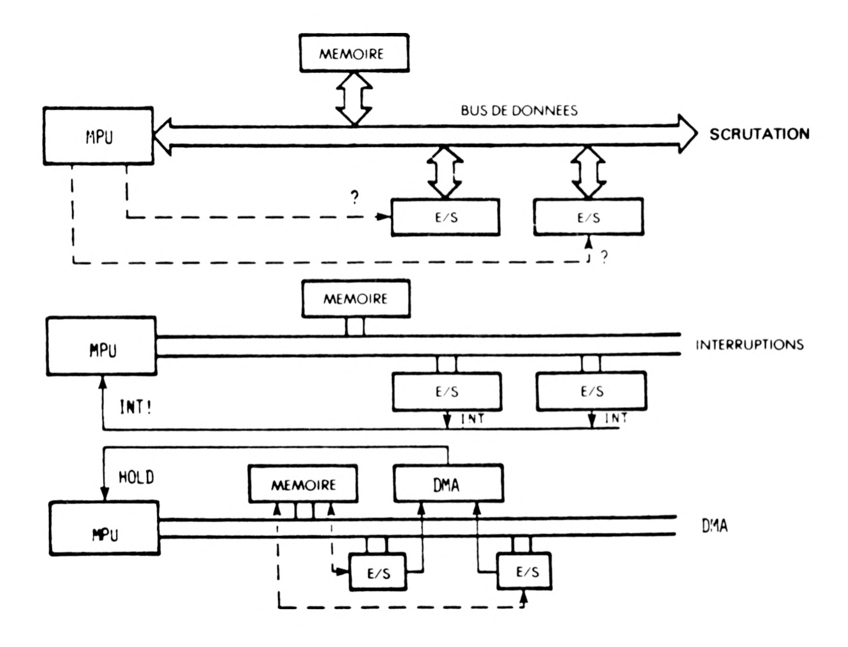


Figure 6.18. — Trois méthodes de gestion des E/S

```
POLL4   IN      A, (ETAT1)      RECUPERER L'ETAT DU
                                PERIPHERIQUE 1
        BIT     7, A            DEMANDE T'IL UN
                                SERVICE ?
        CALL    NZ, UN          SI OUI, LE SERVIR
        IN      A, (ETAT2)      PERIPHERIQUE 2
        BIT     7, A
        CALL    NZ, DEUX
        IN      A, (ETAT3)      PERIPHERIQUE 3
        BIT     7, A
        CALL    NZ, TROIS
        IN      A, (ETAT4)      PERIPHERIQUE 4
        BIT     7, A
        CALL    NZ, QUATRE
        JR      POLL4           PAS DE REQUETE,
                                BOUCLER
```

Nous supposerons que, lorsqu'un périphérique demande un service, le bit 7 de son registre d'état est à « 1 ». Dans ce cas, le programme se branche à la routine de gestion du périphérique, appelée UN pour le périphérique nº 1, DEUX pour le périphérique nº 2, etc...

Il est important de vérifier dans quelle mesure chaque instruction affecte les codes condition. Notons que l'instruction d'entrée ne modifie pas les indicateurs d'état.

Dans certaines architectures de processeurs, les périphériques d'entrées/sorties sont traités, du point de vue de l'adressage, comme des adresses mémoires. C'est ce qu'on appelle l'implantation mémoire des entrées-sorties. Dans ce cas, l'instruction IN aurait été remplacée par une instruction LD, et le reste du programme aurait été modifié.

Les avantages de la méthode d'interrogation sont évidents : elle est simple, ne nécessite aucun hardware spécial et permet la gestion des entrées-sorties en synchronisme avec le déroulement du programme. Son inconvénient est tout aussi évident : la plus grande partie du temps du calculateur est gaspillée en tests destinés à établir si les périphériques nécessitent un service, ou pas. Dans ces conditions, il peut arriver que le processeur s'occupe trop tard d'un périphérique.

Il est donc souhaitable de disposer d'un autre mécanisme, qui garantisse que le processeur soit occupé à des tâches utiles, plutôt qu'à interroger sans succès les périphériques. Cependant, il faut insister sur le fait que la méthode d'interrogation est très largement utilisée lorsque le processeur n'a rien de plus urgent à faire, car elle permet de conserver une organisation très simple au programme. Nous allons maintenant étudier la principale alternative à la méthode d'interrogation : les interruptions.

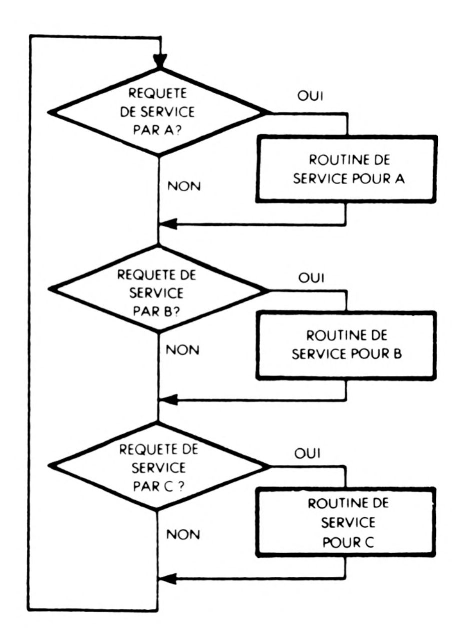


Figure 6.19. — Ordinogramme boucle d'interrogation

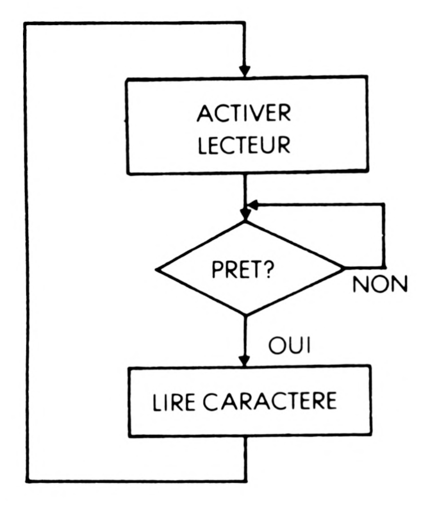


Figure 6.20. — Entrée sur lecteur de rubans

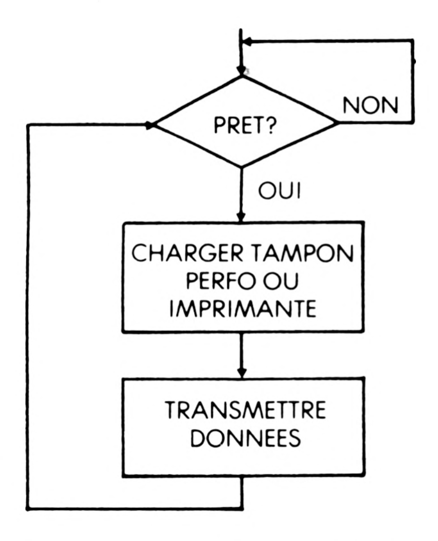


Figure 6.21. — Sortie sur imprimante, ou perforateur de rubans

Les interruptions

Le concept d'interruption est illustré à la figure 6.18. Une ligne spéciale est reliée au microprocesseur : la ligne d'interruption. Plusieurs périphériques d'entrée-sortie peuvent être raccordés à cette ligne. Lorsque l'un

d'entre eux a besoin que le processeur s'occupe de lui, il l'indique sur cette ligne en envoyant une impulsion, ou en maintenant un niveau. Un signal d'interruption est une demande de service, adressée au microprocesseur par un périphérique. Examinons la réponse du processeur à cette demande.

Dans tous les cas, le processeur achève l'instruction en cours, pour éviter toute difficulté de reprise. Il se branche, ensuite, à une routine de traitement de cette interruption, ce qui implique que le contenu du compteur ordinal soit sauvegardé sur la pile. *Une interruption doit donc provoquer la sauvegarde automatique du compteur ordinal sur la pile.* Le registre des indicateurs d'état F doit, lui aussi, être sauvegardé automatiquement, car son contenu a toutes chances d'être modifié par les instructions suivantes. Enfin, si la routine de gestion de l'interruption utilise des registres internes, ces derniers devront aussi être sauvegardés sur la pile.

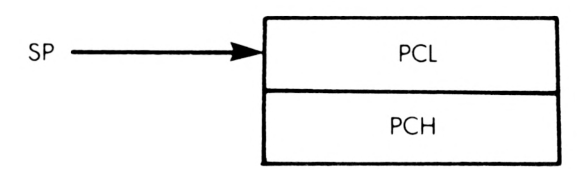


Figure 6.22. — Pile du Z80 après interruption

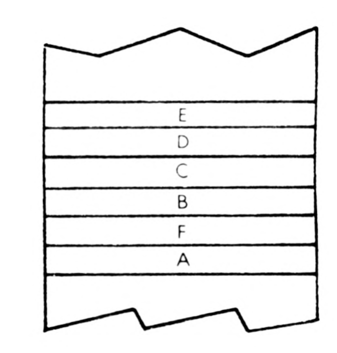


Figure 6.23. — Sauvegarde de quelques registres

Une fois les registres préservés, il est possible de se brancher à la routine de traitement d'interruption appropriée, au terme de laquelle tous les registres devront être restaurés, et une instruction spéciale de retour d'interruption exécutée, pour que le processeur puisse reprendre l'exécution au point où il l'avait interrompue. Examiner, en détail, les lignes d'interruption du Z80.

Les interruptions sur le Z80

Une interruption est un signal, une demande de service, envoyé au microprocesseur. Elle est asynchrone au déroulement du programme. Lorsqu'un programme se branche à un sous-programme, le branchement est dit *synchrone* à l'exécution du programme, puisqu'il est prévu et ordonné par lui. Une interruption, au contraire, est tout à fait impromptue. Elle suspend généralement l'exécution du programme en cours (sans que ce dernier le sache). C'est pour cela qu'elle est dite *asynchrone.*

Trois mécanismes d'interruptions cohabitent sur le Z80 : la demande de bus (BUSRQ, bus request), l'interruption non-masquable (non-maskable interrupt, NMI) et l'interruption ordinaire (INT).

Examinons-les.

La demande de bus

La demande de bus est le mécanisme d'interruption prioritaire du Z80. La séquence d'interruption est présentée à la figure 6.24. En règle générale, aucune interruption ne sera prise en compte avant la fin du cycle-machine en cours. Les interruptions NMI et INT ne pourront, elles, être prises en compte qu'à la fin de l'exécution d'une instruction. Par contre, l'interruption BUSRQ (demande de bus) sera prise en compte dès la fin du cycle-machine en cours, sans attendre la fin de l'instruction. Elle est utilisée pour l'accès-direct mémoire (DMA), et provoque le passage du Z80 en mode DMA (voir notre livre C. 201 pour l'explication des mécanismes d'accès direct mémoire). C'est au terme d'une instruction que si un des deux signaux NMI ou INT est actif, son état est mémorisé par deux bascules internes : la bascule INT et la bascule NMI. En mode DMA, le Z80 suspend son travail, et met son bus de données et son bus d'adresses en état de haute impédance. Ce mode est normalement utilisé par un contrôleur de DMA pour transférer des données, à grande vitesse, entre la mémoire et un périphérique en utilisant le bus d'adresses et le bus de données. La fin d'une opération DMA est indiquée par le retour à l'état inactif du signal BURSQ. A ce point, le Z80 reprend son travail normalement. Il teste notamment si les bascules internes de NMI ou INT sont positionnées, et si oui, il se branche aux routines d'interruption appropriées.

Normalement, l'accès direct mémoire ne concerne pas le programmeur, sauf s'il est soumis à des contraintes de temps importantes. Si un contrôleur de DMA est présent dans le système, le programmeur ne doit pas ignorer que la réponse à une NMI ou une INT peut être retardée par une opération de DMA.

L'interruption non masquable

Elle ne peut pas être empêchée (masquée) par le programmeur. D'où son nom : interruption *non masquable.* Elle est toujours acceptée par le Z80, à

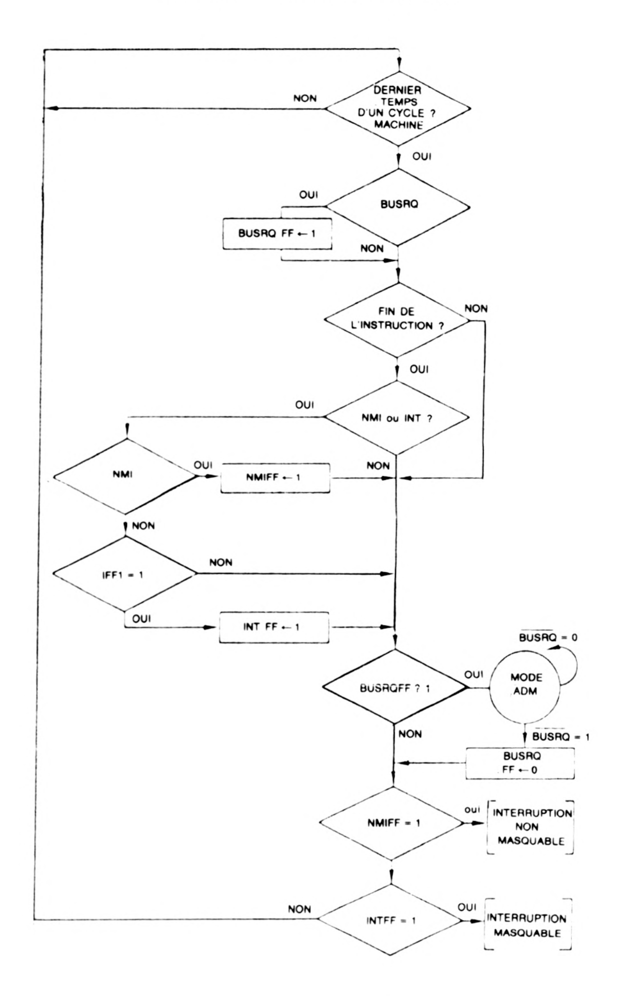


Figure 6.24. — Séquence d'interruption

la fin de l'exécution de l'instruction en cours, en admettant que le signal BUSRQ ne soit pas simultanément actif. (Si une NMI est reçue pendant un BUSRQ, elle positionnera la bascule interne NMI, et sera traitée à la fin du BUSRQ).

L'interruption NMI provoque un empilage automatique du compteur ordinal PC, et un branchement à l'adresse 0066H. Les deux octets représentant cette même adresse sont déposés dans le compteur ordinal. Ils représentent l'adresse du début de la routine de traitement des interruptions non-masquables (voir figure 6.25).

Ce mécanisme d'interruption a été conçu pour être rapide. Il n'est utilisé que dans les cas de très grande urgence. Aussi n'offre-t-il pas la flexibilité de l'interruption masquable décrite plus loin.

Remarquons encore que le programme d'interruption doit être chargé à l'adresse 0066H avant l'utilisation de l'interruption NMI.

L'interruption NMI provoque un branchement automatique à l'adresse 0066H. La séquence des événements est la suivante :

①	PC → pile	(sauvegarde du compteur ordinal)
	IFF1 → IFF2	(sauvegarde de IFF)
	0 → IFF1	(mise à zéro de IFF)
②	SAUT EN 0066H	(exécuter la routine de traitement)

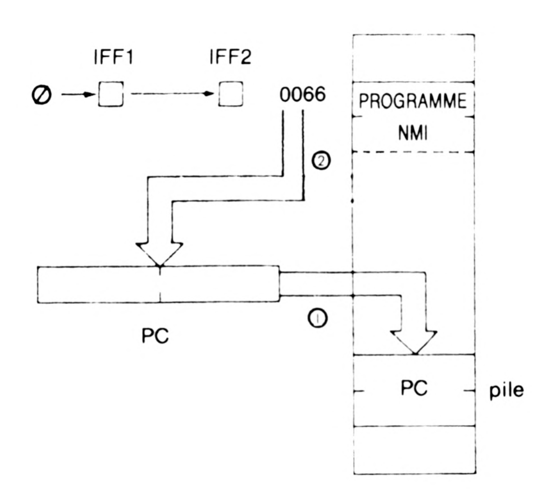


Figure 6.25. — NMI force un vecteur automatique

Au moment de la réception d'une interruption NMI, l'état du masque d'interruption IFF1 est préservé automatiquement dans IFF2. IFF1 est ensuite mis à 0 pour empêcher l'arrivée d'autres interruptions. Cette caractéristique est importante : elle empêche la perte d'interruptions INT, qui sont moins prioritaires et simplifient le hardware extérieur. Le fait qu'une interruption INT soit présente est préservé, de manière interne, par le Z80.

L'interruption NMI est utilisée, normalement, pour des événements très prioritaires tels qu'une horloge temps réel, ou la détection d'une panne d'alimentation.

Le retour de la routine de gestion de l'interruption NMI est assuré par une instruction spéciale, RETN (RETurn from Non maskable interrupt : retour d'interruption non masquable). Le contenu de IFF1 est restauré à partir de IFF2, et le contenu du compteur ordinal PC à partir de la pile. Dans la mesure où le contenu de IFF1 a été mis à 0, lors de la prise en compte de l'interruption NMI, aucune interruption INT n'a pu être accep-

tée pendant le déroulement de la routine NMI : il n'y a pas eu de perte d'information.
Une fois la routine de traitement terminée, les actions entreprises sont :

IFF2 → IFF1 (restaurer IFF)
pile → PC (restaurer le compteur ordinal)

Remarquons que, après la restauration de IFF1, la bascule d'autorisation de l'interruption masquable a été remise dans son état précédent.

Les interruptions ordinaires

L'interruption normale, masquable, INT peut être mise en œuvre dans l'un des trois modes spécifiques du Z80. Le 8080 quant à lui, n'est équipé que d'un seul mode. Elle peut être masquée par le programmeur. Mettre les bascules IFF1 et IFF2 à « 1 » autorise la prise en compte des interruptions INT ; les mettre à « 0 » empêche la détection des INT (elles sont masquées). L'instruction EI permet de mettre ces bascules à « 1 ». L'instruction DI les met à « 0 ». IFF1 et IFF2 sont mis à « 1 » ou à « 0 » simultanément. Pendant l'exécution des instructions EI et DI, les interruptions INT ne sont pas prises en compte, afin d'éviter toute perte d'information.
Examinons maintenant les trois modes d'interruption.

Le mode d'interruption 0

Ce mode est identique au mode unique d'interruption du 8080. Le Z80 travaille en mode 0 lorsqu'il est initialisé (lorsque le signal RESET a été activé), ou lorsque l'instruction IM 0 (Interrupt Mode 0, mode d'interruption 0) a été exécutée. Une fois en mode 0, une interruption INT ne pourra être prise en compte que si la bascule IFF1 est à 1, à condition toutefois qu'un BUSRQ ou une NMI ne soit présente simultanément. L'interruption ne sera détectée qu'à la fin de l'instruction en cours. Le rôle du Z80, lorsqu'il détecte une INT, est d'activer les signaux M1 et IORQ, puis d'attendre.
C'est alors le travail du périphérique interrupteur de reconnaître la présence simultanée de IORQ et de M1 (appellée Acceptation d'INterruption, ou, en anglais, INTA, INTerrupt Acknowledge), et de déposer une instruction sur le bus de données. Le Z80 attend en effet, qu'une instruction soit déposée, au cycle suivant. Généralement, il s'agira d'un RST ou d'un CALL. Ces deux instructions permettent de sauvegarder automatiquement le compteur ordinal sur la pile ; ils provoquent un branchement à une adresse donnée. L'avantage de l'instruction RST est tenir sur un seul octet, donc d'être exécutable rapidement. Son inconvénient est de n'être capable de provoquer le branchement qu'à l'un des 8 emplacements de la page 0 (adresses 0 à 255). L'instruction CALL présente l'avantage d'être une instruction générale de branchement, proposant une adresse sur 16 bits.

Son inconvénient est de tenir sur trois octets, et donc d'être moins rapidement exécutable.

Une fois commencé le traitement d'une interruption, aucune autre interruption ne sera acceptée. IFF1 et IFF2 sont automatiquement mis à « 0 ». La responsabilité du programmeur sera ensuite de mettre une instruction EI (Enable Interrupts : autoriser les interruptions) au bon endroit dans son programme, s'il désire autoriser les interruptions, et de toute façon, avant de revenir de la routine d'interruption.

La séquence détaillée correspondant au mode d'interruption 0 est présentée à la figure 6.26.

Le retour d'une routine de traitement d'interruption est assuré par l'instruction RETI. Rappelons que le programmeur est, parfois, responsable de la désactivation du signal d'interruption du périphérique servi, et dans tous les cas, de restaurer la bascule d'autorisation des interruptions. Le contrôleur de périphériques peut, toutefois, utiliser le signal INTA pour désactiver le signal INT, libérant ainsi le programmeur de cette tâche.

En outre, si la routine d'interruption est chargée de la modification du contenu de certains registres internes, le programmeur doit au préalable les sauver sur la pile avant de les utiliser. Autrement, le contenu de ces registres serait détruit, et lorsque le programme interrompu reprendra la main, il ne pourra être exécuté correctement. Supposons, par exemple, que la routine de traitement des interruptions utilise les registres A,B,C,D,E,H, et L. Elle devra alors, au préalable, les sauver (voir figure 6.27.).

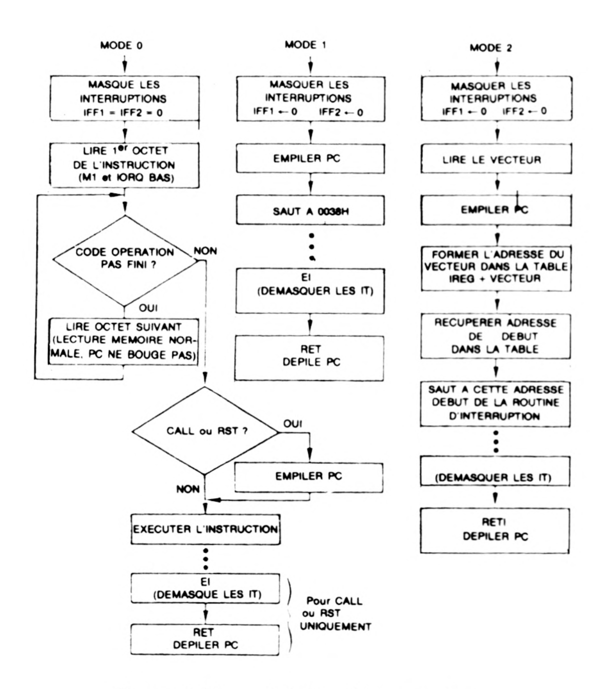


Figure 6.26. — Modes d'interruption

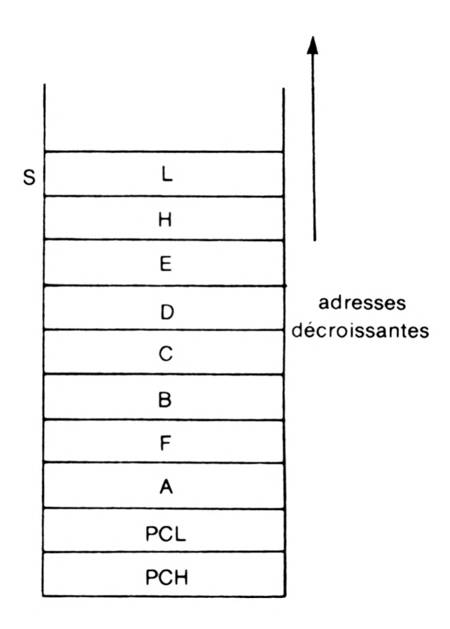


Figure 6.27. — Sauvegarde des registres

Le programme de sauvegarde est :

```
SAVREG      PUSH   AF
            PUSH   BC
            PUSH   DE
            PUSH   HL
```

A la fin de la routine d'interruption, ces registres devront être restaurés. La routine s'achèvera par la séquence d'instructions suivantes :

```
            POP    HL
            POP    DE
            POP    BC
            POP    AF
            EI              (à moins que EI n'ait été utilisé plus
                            tôt)
```

De même, si les registres IX et IY sont utilisés par la routine, ils devront, eux aussi, être préservés, puis restaurés.

Le mode d'interruption 1

Ce mode d'interruption entre en vigueur avec l'exécution de l'instruction IM 1. La prise en compte d'une interruption y provoque un branchement

automatique à l'adresse 0038H, après sauvegarde du compteur ordinal sur la pile (voir figure 6.28). C'est un fonctionnement tout à fait comparable à celui de NMI, à ceci près que l'interruption peut ici être masquée.

Le mode d'interruption automatique, qui provoque un branchement à l'adresse 0038H pour toutes les interruptions, permet de minimiser les circuits externes nécessaires. Inconvénient : il n'autorise le branchement qu'à une *seule* adresse. Le programme situé à l'adresse 0038H est ainsi contraint de déterminer le périphérique interrompant. Ce problème sera évoqué plus loin.

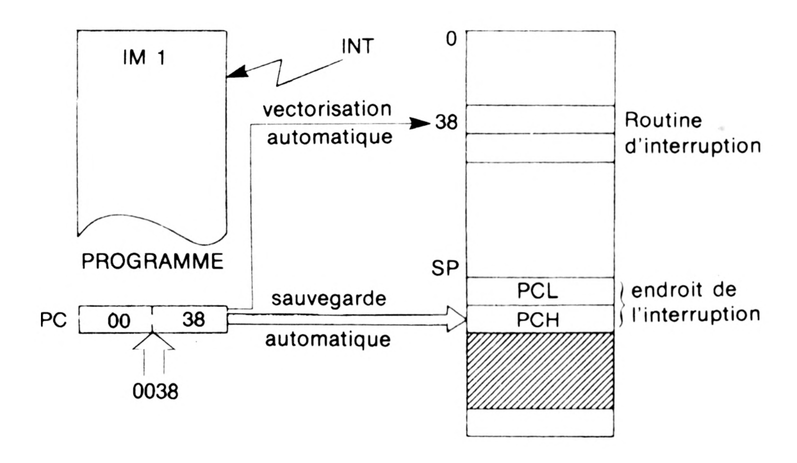


Figure 6.28. — Interruption en mode 1

Une précaution s'impose pour réaliser des transferts programmés d'E/S avec ce mode d'interruption. Le Z80 doit ignorer les données présentes sur le bus pendant le cycle qui suit l'interruption (le cycle de prise en compte de l'interruption, the interrupt acknowledge cycle).

Le mode d'interruption 2 (interruptions vectorisées)

Ce mode entre en vigueur avec l'exécution de l'instruction IM 2. Très puissant, il permet de trouver directement l'adresse de la routine de traitement de l'interruption, grâce à un vecteur d'interruption. Le vecteur d'interruption est une adresse fournie par le périphérique générateur de l'interruption. Elle est utilisée comme pointeur sur l'adresse du début de la routine de traitement de l'interruption. Le mécanisme d'adressage fourni par le mode d'interruption 2 est indirect. Le périphérique interrompant fournit une adresse sur 7 bits, qui est mise à droite des 8 bits du registre I. Un zéro est ensuite ajouté à droite de cet ensemble de 15 bits, fournissant

ainsi une adresse sur 16 bits, qui pointe sur une table, quelque part en mémoire. Chacune d'entre elles est un double-mot, qui est lui-même l'adresse du début d'une routine de traitement d'interruption. (voir figure 6.29 et 6.30).

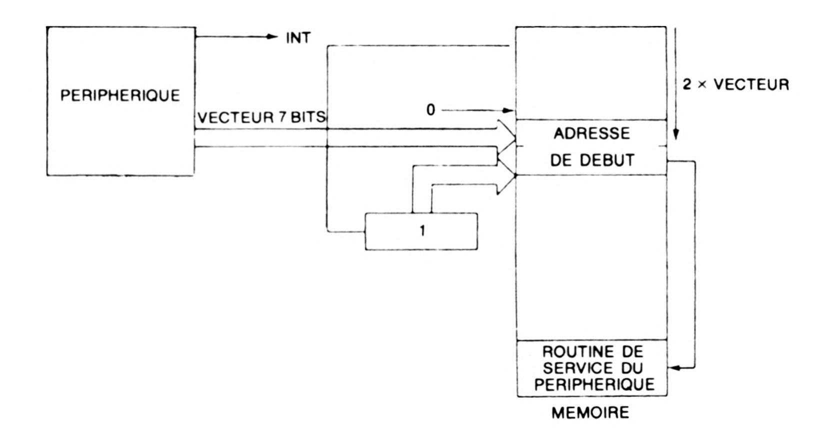


Figure 6.29. — Interruption en mode 2

La table des vecteurs d'interruption peut comporter jusqu'à 128 entrées, d'un double mot chacune.

Dans le mode 2, le Z80 sauve automatiquement le contenu du compteur ordinal sur la pile. L'opération est évidemment nécessaire, puisque le compteur va prendre une nouvelle valeur, prise dans la table des vecteurs d'interruption, à l'entrée correspondant au vecteur fourni par le périphérique interrompant.

Temps de prise en compte d'une interruption

La figure 6.18 présente une comparaison graphique des mécanismes d'interrogation (en haut), et d'interruption (en bas). Il peut arriver qu'avec la méthode d'interrogation, le processeur gaspille beaucoup de temps à attendre que les périphériques soient prêts pour une nouvelle opération.

Avec le mécanisme d'interruptions, le programme est interrompu, l'interruption est traitée, et le programme reprend. Inconvénient évident : le recours obligatoire à plusieurs instructions supplémentaires, au début et à la fin. Ce qui introduit un délai, appelé en anglais « overhead », entre l'interruption et l'exécution de la première instruction de la routine de traitement de l'interruption.

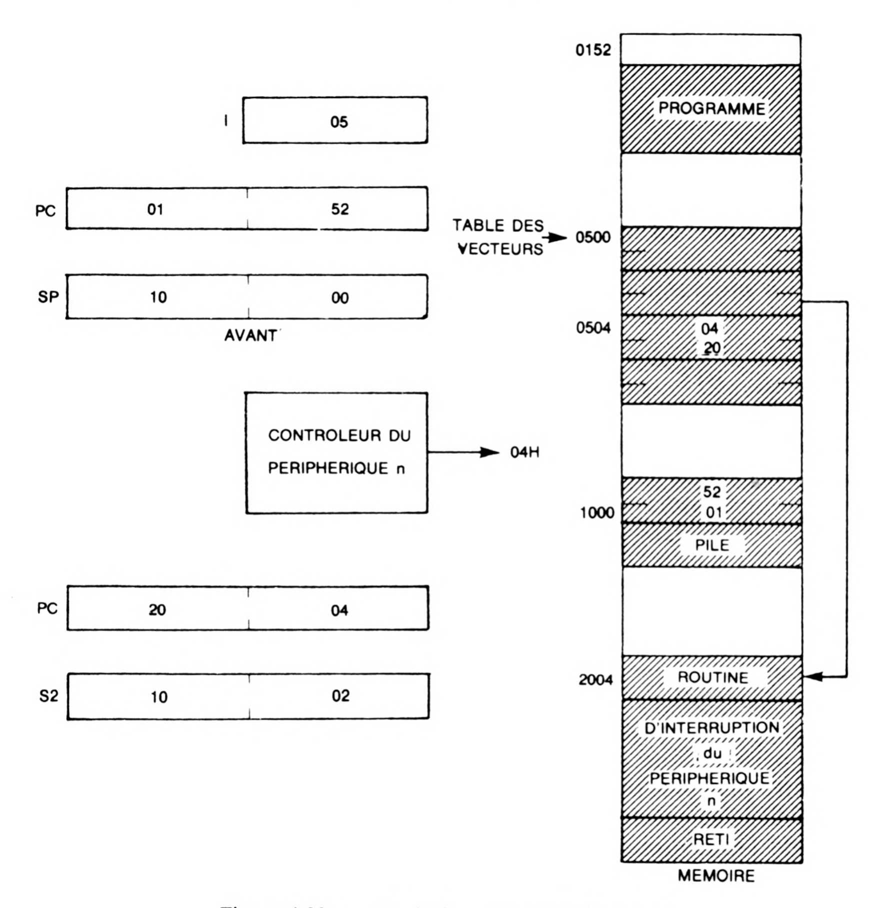


Figure 6.30. — Mode 2 — un exemple concret

Exercice 6.28 : *En vous servant de la table indiquant le nombre de cycles utilisés par chaque instruction (en appendice), calculer le temps perdu à sauver, et à restaurer, les registres A,B,D,H.*

Le fonctionnement des lignes d'interruption est maintenant éclairci. Restent en suspens deux problèmes importants :

1. Que se passe-t-il lorsque deux périphériques interrompent en même temps le processeur ?
2. Que se passe-t-il si une interruption a lieu, alors que la routine de traitement d'une autre interruption est en cours ?

Plusieurs périphériques connectés à la même ligne

Lorsqu'une interruption se produit, le processeur se branche à une certaine adresse. Avant qu'un traitement effectif puisse avoir lieu, la routine

de traitement des interruptions doit déterminer quel périphérique a demandé l'interruption. Comme d'habitude, deux méthodes sont pour cela possibles : une méthode programmée et une méthode hardware.

La méthode programmée consiste à interroger successivement chaque périphérique, et à lui demander : « Est-ce toi qui a activé le signal d'interruption ? ».

Si la réponse est négative, le périphérique suivant est, à son tour, questionné. Ce processus est illustré par la figure 6.31.

Voici un programme exemple :

```
POLINT  IN    A, (ETAT1)   LIRE L'ETAT DU PERIPHE-
        BIT   7, A         RIQUE
        JP    NZ, UN       A-T-IL ACTIVE LE SIGNAL
        IN    A, (ETAT2)   INT ?
        BIT   7, A         SI OUI, LE SERVIR
        JP    NZ, DEUX     SINON TESTER LE
        etc..              SUIVANT
```

La méthode hardware utilise des composants supplémentaires, mais fournit l'adresse de la routine de traitement de l'interruption, en même temps qu'elle active le signal INT. Le circuit utilisé désormais universellement pour cette opération s'appelle un « PIC » (de l'anglais Priority Interrupt Controller : gestionnaire des priorités d'interruption). Un PIC place automatiquement sur le bus de données l'adresse de branchement associée périphérique qui a interrompu.

Pour être plus plus précis, le PIC fournira, en mode 0, un RST (un octet), ou un CALL (3 octets), sur le bus de données, en réponse à la prise en compte de l'interruption par le Z80 (INTA). Cette prise en compte rend possible un branchement direct à la routine d'interruption, et minimise ainsi le supplément de temps exigé par la recherche de l'adresse de cette routine.

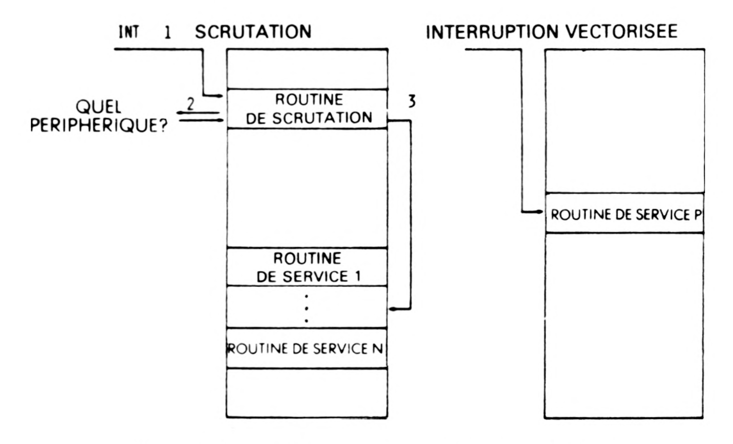


Figure 6.31. — Interrogation contre vectorisation

A noter qu'il est nécessaire d'envoyer une instruction d'appel de sous-programme, puisque le Z80, en mode 0, ne sauve pas automatiquement le compteur ordinal sur la pile.

Dans la plupart des cas, la vitesse de réaction à une interruption n'est pas cruciale : la méthode d'interrogation est alors employée. Lorsque le temps de réponse est primordial, la méthode hardware est préférée.

Interruptions simultanées

Autre problème possible : un périphérique active la ligne d'interruption pendant le déroulement d'une routine de traitement des interruptions. Voyons ce qui se passe dans ce cas, et de quelle manière l'usage de la pile permet d'apporter une solution. Nous avons affirmé, au chapitre 2, qu'il s'agit là de l'un des rôles essentiels d'une pile. Nous allons maintenant le démonter, en nous référant à la figure 6.33. Sur cette figure, le temps s'écoule de gauche à droite. Les contenus de la pile sont représentés dans la partie inférieure. Sur la gauche, nous voyons qu'au temps T0 le programme P est exécuté. En nous déplaçant sur la droite, nous rencontrons l'interruption I1, au temps T1. Nous supposerons que le masque d'interruptions est dans la position « interruptions autorisées », permettant ainsi à l'interruption I1 d'être prise en compte. Le programme P est suspendu. Nous voyons, en bas de l'illustration, que la pile contient maintenant le compteur ordinal et le registre d'état du programme P, ainsi peut être que d'autres registres, sauvés par la routine d'interruption, ou par I1 elle-même.

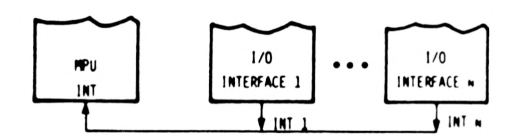


Figure 6.32. — Plusieurs périphériques peuvent utiliser la même ligne d'interruption

Au temps T1, l'exécution de la routine d'interruption associée à I1 commence. Au temps T2, l'interruption I2 arrive. Nous supposerons que la seconde a une priorité supérieure à la première. Dans le cas contraire, elle serait ignorée jusqu'à la fin du traitement de I1. A T2, les registres associés à I1 sont sauvés, comme nous le voyons en bas de la figure. De nouveau, les contenus du compteur ordinal et de AF sont empilés. La routine de traitement de I2 peut aussi décider de sauver d'autres registres. La routine d'interruption associée à I2 va maintenant être exécutée jusqu'au bout, au temps T3.

Au terme de I2, le contenu de la pile est automatiquement dépilé dans les registres du Z80 [partie inférieure de la figure 6.33.]. Ainsi, l'exécution de

I1 reprend automatiquement. Malheureusement, au temps T4, une interruption de priorité supérieure arrive. La figure montre que les registres de I1 sont à nouveau empilés. L'interruption I3 est exécutée de T4 à T5, où elle s'achève. A ce moment, le contenu de la pile est dépilé vers les registres du Z80. L'interruption I1 reprend, jusqu'au bout cette fois, en T6. Les registres précédemment empilés sont, à nouveau, dépilés, et le programme P peut continuer. Le lecteur vérifiera qu'à ce moment, la pile est de nouveau vide. Les lignes pointillées symbolisant les suspensions de programme indiquent, en part, le nombre de niveaux de la pile.

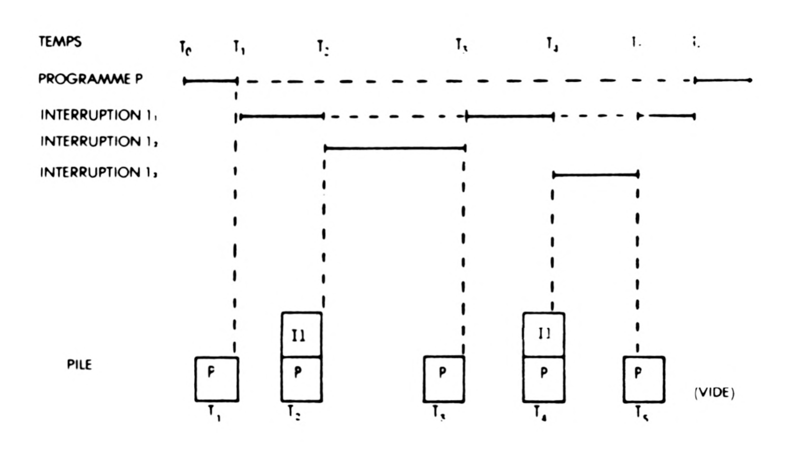


Figure 6.33. — Contenu de la pile lors d'interruptions multiples

Exercice 6.29 : *Supposons que la taille disponible pour la pile d'un programme soit limitée à 300 emplacements mémoire ; que tous les registres doivent être sauvés à chaque interruption ; et que le programmeur autorise les interruptions imbriquées, c'est-à-dire qu'une nouvelle interruption puisse interrompre une routine de traitement d'interruption. Quel est, dans ce cas, le nombre maximum d'interruptions « simultanées » qui puisse être traité ? D'autres facteurs peuvent-ils contribuer à réduire encore le nombre maximum possible d'interruptions simultanées ?*

Insistons sur le fait qu'un système microprocesseur est, en général, connecté à un nombre limité de périphériques utilisant les interruptions. Il est donc peu probable que de nombreuses interruptions simultanées puissent avoir lieu dans ce système.

Nous avons maintenant résolu tous les problèmes habituellement associés aux interruptions. L'utilisation de ces dernières est, en fait, simple. Même le programmeur débutant pourra s'efforcer d'en tirer profit.

EN RÉSUMÉ

Nous venons de voir l'ensemble des techniques utilisées pour communiquer avec le monde extérieur. Depuis les routines élémentaires d'entrée-sortie, jusqu'aux programmes plus complexes de communications avec des périphériques réels. Nous avons appris à écrire tous les types de programmes, et même à évaluer l'efficacité d'un ensemble de programmes de tests, dans le cas de transferts parallèles et de la conversion parallèle-série. Nous savons désormais organiser le travail de plusieurs périphériques, au moyen des techniques d'interrogation et d'interruption. Naturellement, de nombreux périphériques moins usuels peuvent être connectés à un système. A l'aide de la palette des techniques présentées, et d'une bonne compréhension des périphériques que vous désirez utiliser, vous devriez être à même de résoudre la plupart des problèmes courants.

Au chapitre suivant, nous examinerons les caractéristiques des périphériques réels habituellement connectés au Z80. Puis, nous passerons en revue les structures de données de base utilisables par le programmeur.

Exercice 6.30 : *Nous définirons le « coût » d'une interruption comme étant le temps total impliqué par celle-ci, non comprises les instructions explicitement nécessaires à son traitement. Calculer ce coût, en supposant que le travail ait lieu en mode d'interruption 0, que tous les registres doivent être sauvés, et qu'une instruction RST soit envoyée en réponse à l'acceptation de l'interruption.*

Exercice 6.31 : *Un afficheur DEL 7 segments permet d'afficher d'autres symboles que les chiffres hexadécimaux. Calculer les codes de H,I,J,O,P,S,U,Y, g,h,i,j,l,n,o,p,r,t,u,y.*

Exercice 6.32 : *Un organigramme de traitement des interruptions est présenté à la figure 6.34.*
Répondez aux questions suivantes :

A. *Quelle partie du traitement est réalisée par hardware ? quelle autre par programme ?*
B. *Quelle est l'utilité du masque ?*
C. *Combien faut-il sauvegarder de registres ?*
D. *Comment identifie-t-on le périphérique interrompant ?*
E. *Que fait l'instruction RETI ? En quoi diffère-t-elle de l'instruction RET ?*
F. *Suggérez un mode de traitement dans le cas de débordement des limites permises pour la pile.*
G. *Quel est le « coût » (voir exercice 6.30) induit par le mécanisme d'interruptions ?*

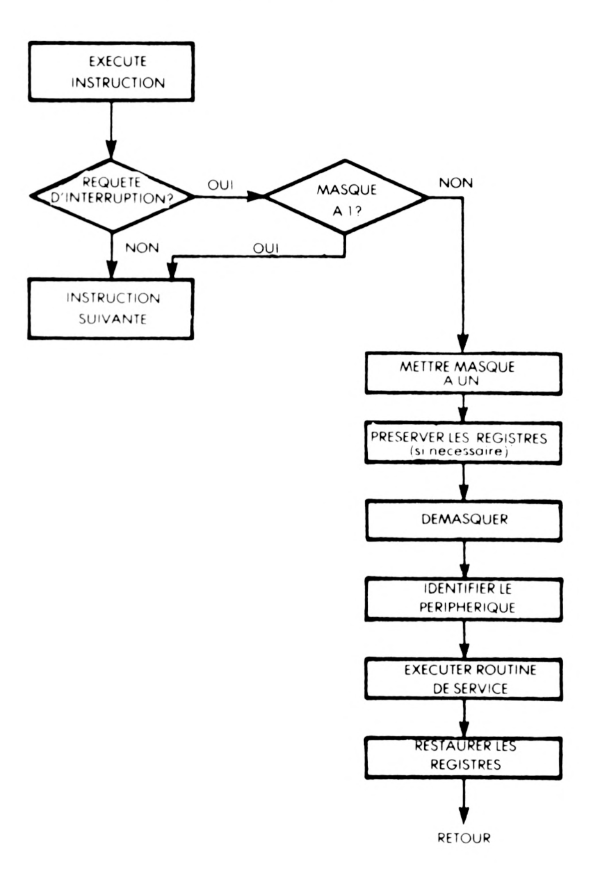


Figure 6.34. — Logique de l'interruption

7

LES PÉRIPHÉRIQUES D'ENTRÉE/SORTIE

INTRODUCTION

Nous avons appris à programmer le Z80 dans la plupart des situations usuelles. Il faut toutefois faire mention des circuits périphériques normalement connectés au microprocesseur. En raison des progrès de l'intégration LSI (de l'anglais Large Scale Integration : intégration à grande échelle), de nouveaux circuits ont été introduits. Si, évidemment, la programmation d'un système implique toujours, et d'abord, celle du microprocesseur lui-même, elle nécessite aussi, désormais, *celle des différents circuits d'entrée-sortie.* En fait, il est souvent plus difficile de se souvenir de la manière de programmer les différentes options de contrôle d'un circuit d'entrée/sortie, que de programmer le microprocesseur ! Ce n'est pas que la programmation, en elle-même, soit plus difficile, mais simplement que chaque circuit a ses propres caractéristiques. Nous commencerons par étudier le plus général des périphériques d'entrée-sortie, le circuit d'entrée-sortie programmable : le PIO (de l'anglais Programmable Input-Output), puis nous verrons quelques circuits d'entrée/sortie Zilog.

Le « PIO standard »

Il n'existe pas, en réalité, de PIO standard, mais tous se ressemblent plus ou moins. La fonction d'un PIO est de fournir plusieurs ports pour la connexion des périphériques d'entrée-sortie. (Un « port » est simplement un ensemble de 8 lignes d'entrée-sortie). Chaque PIO possède, au moins, deux ensembles de 8 lignes, et requiert un *« tampon » de données* (en anglais = data buffer) pour stabiliser le contenu du bus de données au moins en sortie. Il sera donc équipé d'au moins un tampon pour chaque port.

On sait que le microprocesseur utilise un *protocole,* ou des interruptions, pour communiquer avec un périphérique d'entrée-sortie (en abrégé : E/S). Le PIO en fait de même, et doit donc posséder au moins *deux lignes de contrôle par port* pour établir le dialogue avec le périphérique qui lui est connecté.

Le microprocesseur devra pouvoir lire l'état de chaque port. Chaque port comprendra donc un ou plusieurs *bits d'état* (status bits). Enfin, chaque PIO comportera un certain nombre d'options, suivant l'utilisation qui sera faite de ses ressources. Le programmeur devra avoir accès à un registre spécial du PIO, afin de lui préciser les options qu'il choisit. C'est *le registre de commande* (control register, CR). Dans certains cas, les informations d'état font partie du registre de commande.

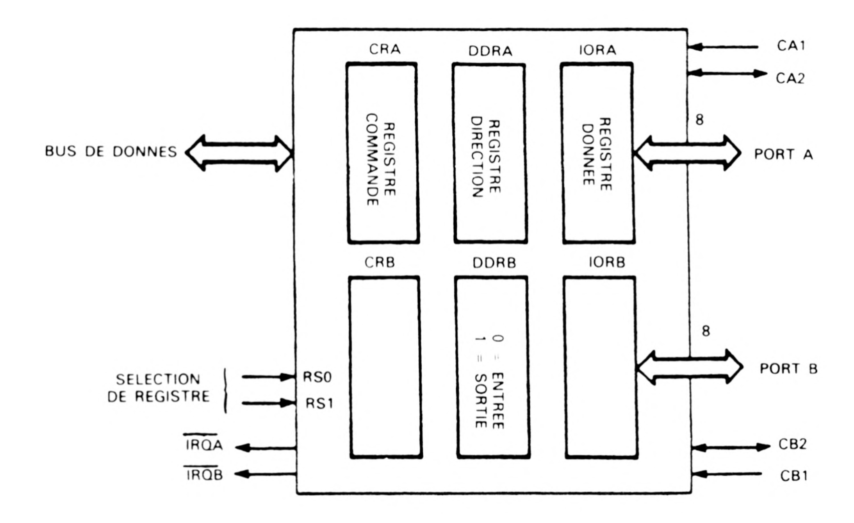


Figure 7.1. — PIO typique

Une propriété essentielle du PIO est que chaque ligne doit pouvoir être utilisée soit en entrée, soit en sortie. Un diagramme apparaît à la figure 7.1. Le programmeur a la possibilité de préciser les lignes qu'il désire utiliser respectivement en entrée et en sortie. Pour programmer la direction des lignes, chaque port possède un *registre de direction de données* (data direction register, DDR), Un « 1 » dans un bit du registre de direction de données indique que la ligne correspondante est en entrée. Un « 0 » indique une sortie. Cette correspondance est utilisée en raison de l'analogie anglaise entre Input et 1, et Output et 0.

Lorsque le système est mis sous tension, il est très important que toutes les lignes soient configurées en *entrée.* Si, en effet, le microprocesseur est

connecté à un périphérique dangereux, il risquerait, dans le cas contraire, de l'activer accidentellement. Après l'initialisation (reset), tous les registres sont normalement mis à 1, et par voie de conséquence, toutes les lignes en entrée. La connexion au microprocesseur apparaît sur la gauche de l'illustration. Le PIO se connecte naturellement au bus de données 8 bits, au bus d'adresses et au bus de contrôle. Le programmeur a simplement à charge de préciser l'adresse des registres du PIO auquel il veut avoir accès.

Le registre interne de commande

Le registre de commande du PIO permet de choisir différentes options de fonctionnement, de générer ou de tester des interruptions, et de fournir un protocole d'échange automatique. Une description complète de ses possibilités ne s'impose pas ici. Simplement, l'utilisateur de n'importe quel système utilisant un PIO devra se référer aux documentations stipulant le mode de positionnement des différents bits du registre de commande. Chaque fois que le système est initialisé, le programmeur doit charger dans le registre de commande les valeurs correspondant à l'utilisation souhaitée du PIO.

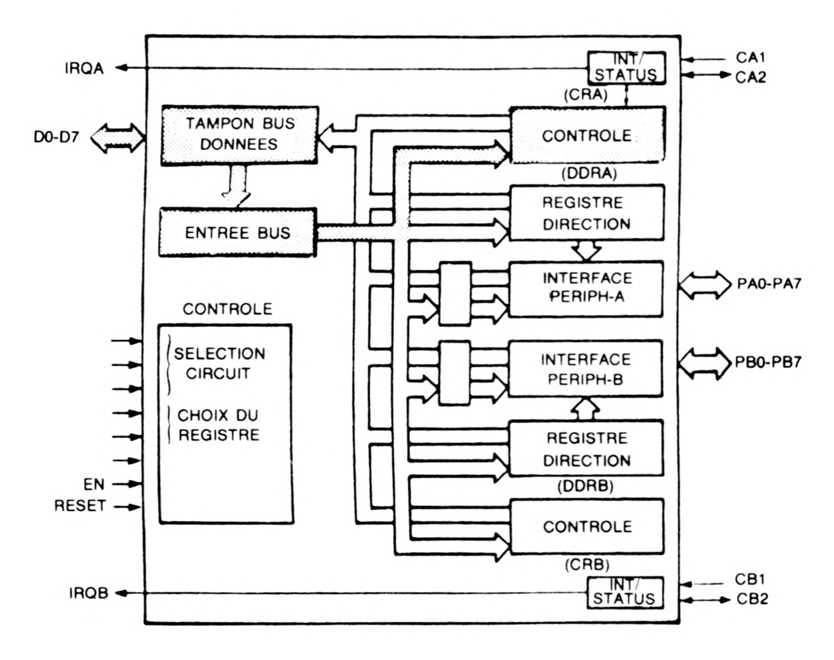


Figure 7.2. — PIO = Charger le registre de commande

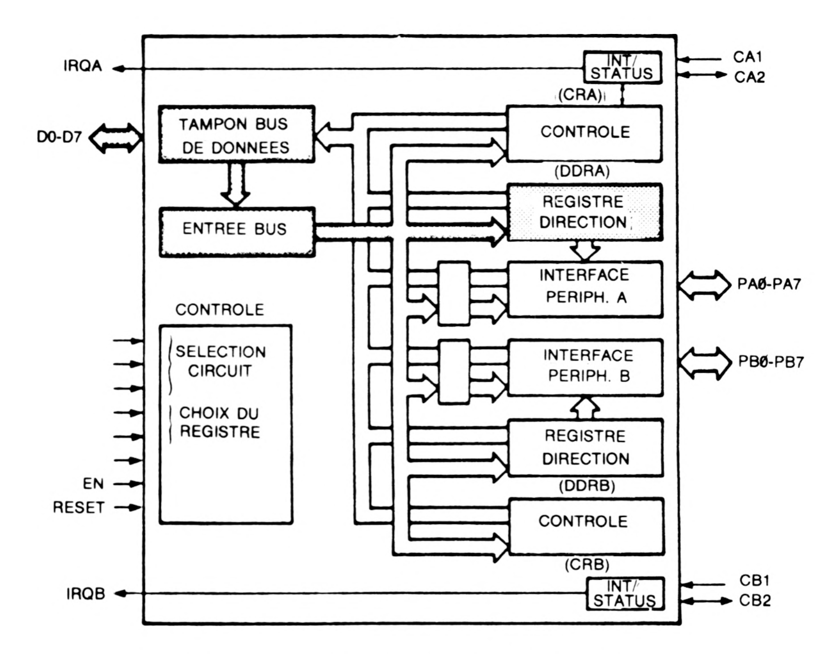


Figure 7.3. — PIO = Charger le registre de direction

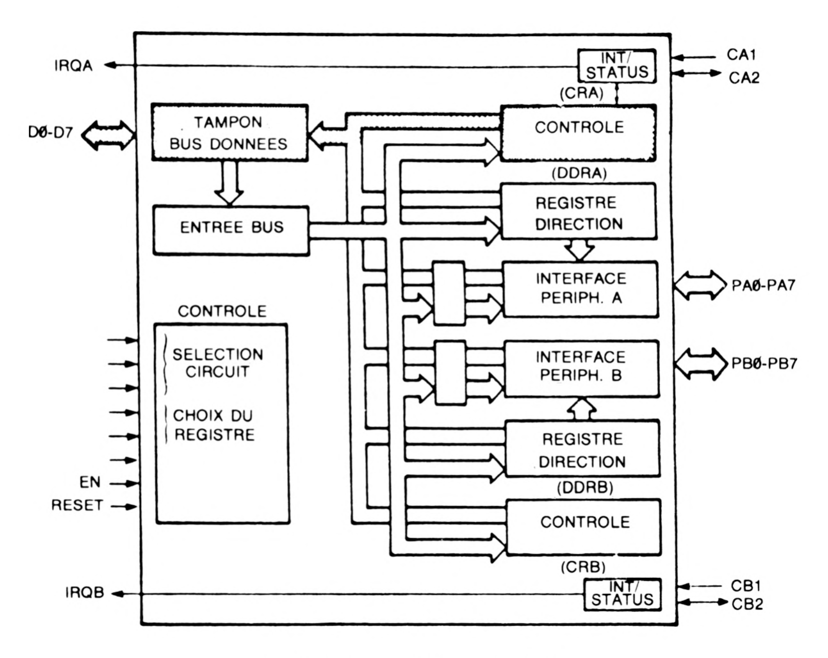


Figure 7.4. — PIO = Lire l'état

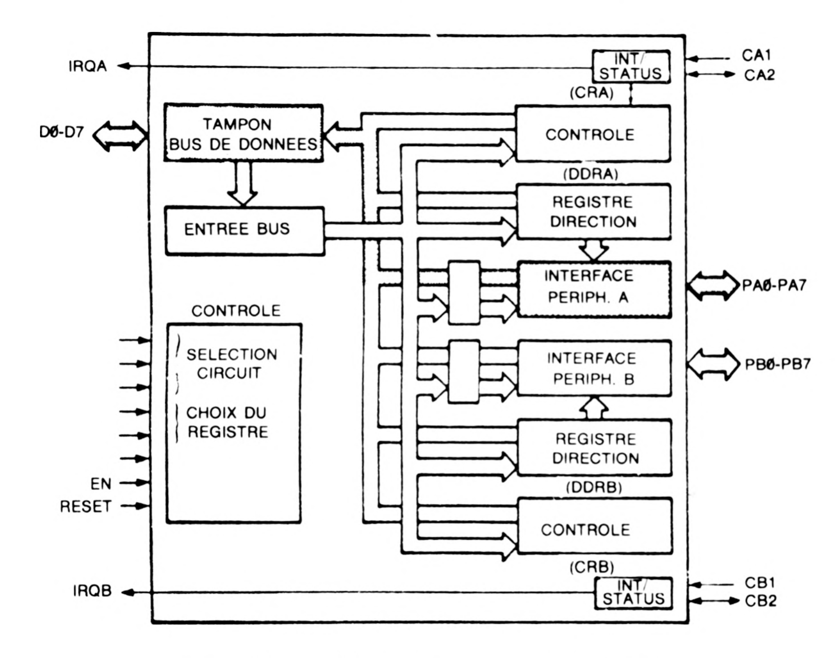


Figure 7.5. — PIO : Lecture d'un octet de données

Programmation d'un PIO

Voici une séquence typique de programmation du canal d'un PIO en entrée :

Charger le registre de commande

Il faut, pour cela, recourir à un transfert de données entre un registre du Z80 (généralement l'accumulateur), et le registre de commande du PIO. Les options choisies et le mode de fonctionnement du PIO sont ainsi positionnés (cf. figure 7.2). Cette opération n'a normalement lieu qu'une seule fois, au début du programme.

Charger le registre de direction

La direction dans laquelle on veut utiliser chaque ligne est ainsi spécifiée. (cf. figure 7.3).

Lecture du registre d'état

La valeur de ce registre indique si un octet est disponible, ou non, sur les lignes d'entrée (cf. figure 7.4).

Lecture du port

L'octet est lu dans le Z80 (voir figure 7.5).

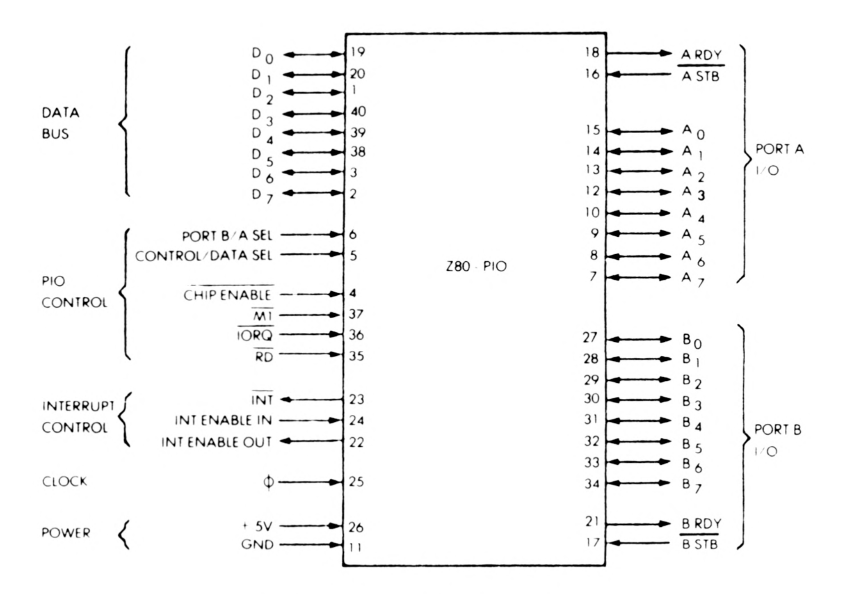


Figure 7.6. — Brochage du PIO Z80

Le Z80 PIO de Zilog

Le Z80 PIO comporte deux ports, dont l'architecture est comparable à celle du modèle standard précédemment décrit. Son brochage est présenté à la figure 7.6, et un schéma de son organisation interne à la figure 7.7.

Chaque port du PIO contient six registres : un registre d'entrée de 8 bits, un registre de sortie de 8 bits, un registre de choix du mode de fonctionnement de 2 bits, un registre masque de 8 bits, un registre de direction de données de 8 bits et un registre de 2 bits contrôlant le fonctionnement du registre masque.

Le PIO peut fonctionner dans l'un des quatre modes sélectionnables à l'aide du registre de choix de mode [2 bits]. Ces quatre modes sont les suivants : sortie octet, entrée octet, bidirectionnel et bit.

Les deux bits des registres de contrôle du masque sont chargés par le programmeur ; ils précisent s'il faut surveiller l'état haut ou bas des lignes du périphérique, et énoncent les conditions de génération d'une interruption.

Le registre 8 bits de sélection de direction des lignes fait travailler chaque ligne en entrée ou en sortie, lorsqu'on travaille dans ce mode.

Programmation du PIO Zilog

Une séquence typique de programmation du PIO, par exemple en mode bit, est la suivante :

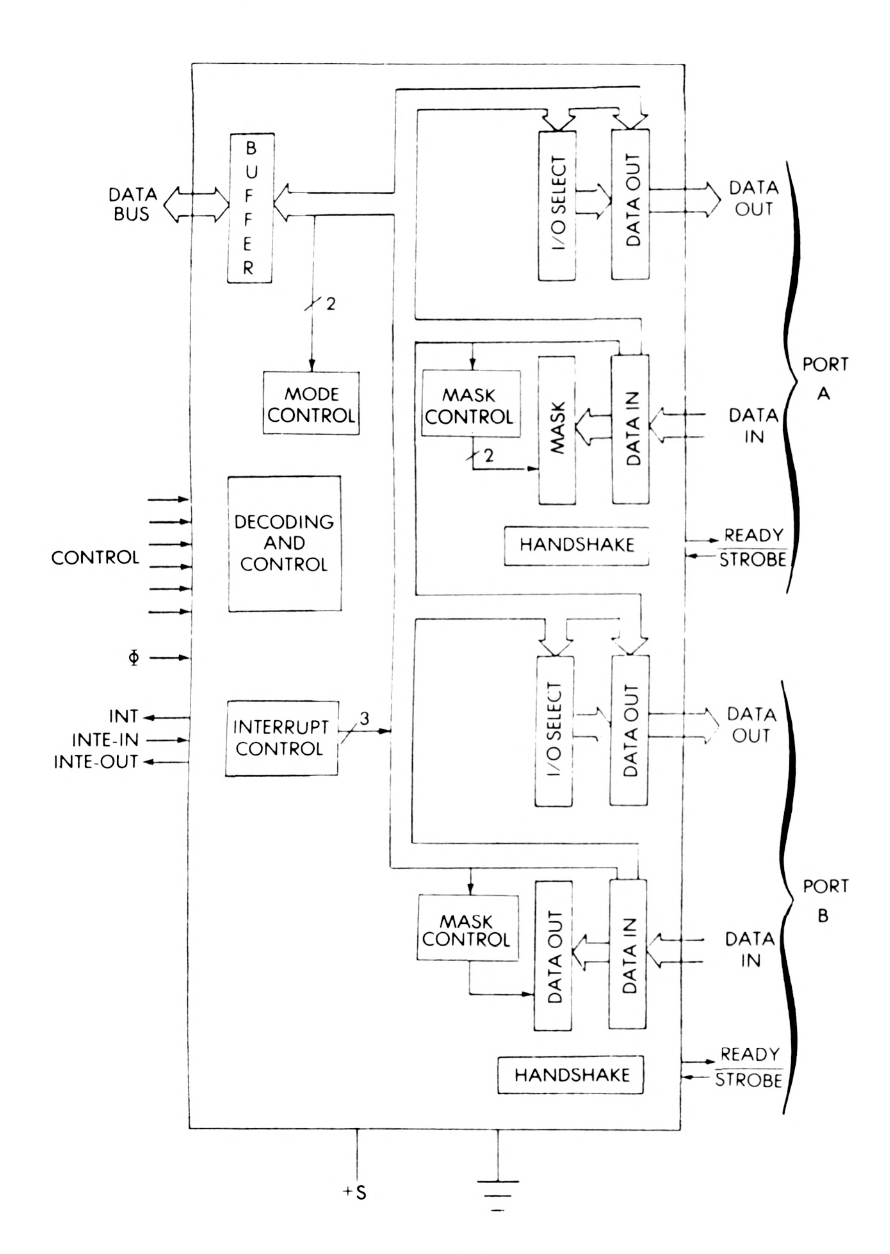


Figure 7.7. — Schéma fonctionnel du Z80 PIO

Chargement du registre de choix de modes, pour spécifier le mode bit.

Chargement du registre de sélection de direction du port A, pour indiquer que les lignes 0-5 seront des entrées, et les lignes 6 et 7 des sorties.

Le contenu du port peut ensuite être lu à l'aide d'une instruction d'entrée spécifiant l'adresse du PIO.

Le registre masque pourra être employé à la surveillance des conditions particulières d'état.

Le lecteur intéressé par une description complète du fonctionnement du PIO est invité à se reporter au livre qui fait suite au présent ouvrage, dans notre série, *le livre d'application du Z80.*

Le Z80-SIO

Le SIO (de l'anglais Serial Input-Output : Entrées-Sorties Série) est un circuit périphérique destiné à faciliter les communications série en mode asynchrone. Il comprend une UART, c'est-à-dire un émetteur-récepteur asynchrone universel (de l'anglais Universal Asynchronous Receiver Transmitted). Sa fonction essentielle est d'effectuer les conversions parallèle-série et série-parallèle. En plus, ce circuit dispose de possibilités très sophistiquées, telles la gestion automatique de protocoles orientés octet complexes.

Il peut aussi fonctionner en mode synchrone, comme une USRT, et aussi tester et générer des CRC. Il permet un choix du mode d'appel, d'interruption et de transfert de blocs. La description complète de ce circuit dépasse le propos de ce livre d'introduction. Elle figure dans notre *livre d'applications du Z80.*

Autres circuits d'E/S

Le Z80 est souvent utilisé en remplacement du 8080. Il a donc été conçu pour être associé à l'un quelconque (ou presque) des circuits périphériques du 8080, aussi bien qu'aux circuits d'E/S spécifiques fabriqués par Zilog. En pratique, tous les périphériques du 8080 peuvent être utilisés dans un système construit autour d'un Z80.

CONCLUSION

Pour utiliser efficacement les circuits d'entrée-sortie, il est nécessaire de comprendre, dans le détail, le fonctionnement de chaque bit [ou groupe de bits] des différents registres de commande. Ces nouveaux circuits complexes permettent de prendre en charge automatiquement des procédures qui, auparavant, devaient l'être par programme, ou par de logique spécialisée. En particulier, dans des composants tels que le SIO, une grande partie des protocoles d'échange est automatisée. De même, la gestion des interrup-

tions devrait permettre au lecteur de comprendre les fonctions des principaux signaux et registres. Naturellement, de nouveaux composants continuent d'apparaître sur le marché, prenant en charge des algorithmes de plus en plus compliqués.

8

EXEMPLES D'APPLICATION

INTRODUCTION

Ce chapitre présente une collection de programmes utilitaires, destinés à mettre à l'épreuve vos nouvelles connaissances en programmation. Ces programmes [ou « routines »] se rencontrent dans de nombreuses applications, et sont généralement appelés « routines utilitaires ». Ils exigent une synthèse des connaissances et des techniques présentées jusqu'ici.

Nous allons présenter des programmes capables de lire des caractères depuis des circuits d'E/S, et de les traiter de différentes façons. Mais d'abord, nous allons apprendre à mettre à zéro une zone de mémoire. (Ce n'est pas toujours nécessaire. Chaque programme est présenté uniquement à titre d'exemple).

MISE À ZÉRO D'UNE ZONE DE MÉMOIRE

Nous voulons mettre à zéro le contenu des emplacements mémoire de l'adresse BASE à l'adresse BASE + LONGUEUR, LONGUEUR étant inférieure à 256.

Voici le programme :

```
MISE-A-ZERO  LD   B, LONGUEUR    LONGUEUR DANS B
             LD   A, 0
             LD   HL, BASE       POINTE SUR LA ZONE
NETTOYER     LD   (HL), A        0 DANS LA CASE
                                 MEMOIRE
             INC  HL             POINTE SUR CASE
                                 SUIVANTE
             DEC  B              DECREMENTE LE
                                 COMPTEUR
             JR   NZ, NETTOYER   FIN DE LA ZONE ?
             RET
```

La longueur de la zone mémoire est ici supposée être égale à LONGUEUR. La paire HL pointe, à chaque fois, sur la case à « nettoyer » (mettre à 0). Le registre B, comme d'habitude, est utilisé comme compteur.

L'accumulateur A est chargé, une seule fois, avec la valeur 0, puis copié dans les cases mémoires successives.

Dans un programme de test du fonctionnement de la mémoire, cette routine utilitaire pourrait, par exemple, mettre à zéro un bloc de mémoire, qui serait relu ultérieurement, en testant que tous les mots sont restés à 0.

Le programme ci-dessus est la traduction directe de l'idée que l'on se fait d'une routine de mise à zéro. Nous l'améliorerons de la manière suivante :

```
MISE-A-ZERO  LD    B, LONGUEUR
             LD    HL, BASE
BOUCLE       LD    (HL), 0
             INC   HL
             DJNZ  BOUCLE
             RET
```

Les deux améliorations ont été obtenues en éliminant l'instruction LD A, 0, remplacée par un chargement direct de la valeur « zéro » dans la case pointée par le registre HL, et en utilisant l'instruction spéciale DJNZ.

Cet exemple montre que, *chaque fois qu'un programme est écrit, même s'il est correct, il est généralement possible de l'améliorer, après examen attentif.* Une bonne connaissance du jeu complet d'instructions est, pour cela, nécessaire. Ces dernières ne sont pas faites simplement pour le plaisir. Elles permettent de réduire le temps d'exécution du programme, le nombre d'instructions, donc l'espace mémoire utilisé, et aussi, généralement, d'améliorer la lisibilité du programme, donc ses chances d'être correct.

Exercice 8.1 : *Ecrire un programme de test mémoire qui mette à zéro une zone mémoire de 256 octets, puis vérifie que chaque case mémoire est bien à zéro. Ce programme écrira ensuite la valeur « 11111111 » dans toutes les cases de la zone, et effectuera la vérification. Il écrira la valeur « 01010101 », et effectuera une nouvelle vérification. Enfin, il écrira la valeur 10101010 dans tous les mots de la zone, et en vérifiera le contenu.*

Exercice 8.2 : *Modifier le programme ci-dessus pour qu'il écrive alternativement les valeurs « 00000000 » et « 11111111 » dans la zone mémoire.*

Interrogeons, maintenant, des périphériques pour savoir s'il est temps de s'occuper d'eux.

INTERROGATION DE CIRCUITS D'E/S

Nous supposerons que ces circuits d'E/S sont connectés à notre système. Leurs registres d'état sont situés aux adresses périphériques ETAT1, ETAT2, ETAT3.

Voici le programme :

```
TEST    IN    A, (ETAT1)     LIRE ETAT CIRCUIT 1
        BIT   7, A           TESTE LE BIT « PRET » (BIT 7)
        JP    NZ, TROUVE1    SAUT AU PROGRAMME DE
                             TRAITEMENT 1 SI PRET
        IN    A, (ETAT2)     IDEM POUR CIRCUIT 2
        BIT   7, A
        JP    NZ, TROUVE2
        IN    A, (ETAT3)     IDEM POUR CIRCUIT 3
        BIT   7, A
        JP    NZ, TROUVE3
```

ici se placent les instructions traitant le cas où aucun circuit n'est prêt...

Après l'instruction BIT, l'indicateur Z sera mis à 0 si le bit 7 de ETAT n'est pas zéro. L'instruction JP NZ (saut si pas zéro) provoquera alors le branchement à la routine TROUVE appropriée. Souvenons-nous que l'indicateur Z prend la valeur 1 (condition zéro = vraie) à la suite d'opérations laissant un résultat nul.

LECTURE DE CARACTÈRES

Supposons que nous venions de nous rendre compte qu'un caractère est disponible sur le clavier. Nous lirons les caractères provenant du clavier, et les déposerons dans une zone mémoire, appelée TAMPON, jusqu'à la rencontre d'un caractère spécial appelé SPC, dont le code a été défini au préalable.

Le sous-programme LIRE-CAR lit un caractère en provenance du clavier (voir chapitre 6 pour plus de détails), et le laisse dans l'accumulateur. Nous supposerons qu'il est possible de lire au maximum 256 caractères avant que le caractère SPC soit trouvé.

```
CHAINE   LD    HL, TAMPON   POINTE SUR LA ZONE
SUIVANT  CALL  LIRE-CAR     LIT UN CARACTERE
         CP    SPC          TESTE SI C'EST LE CARAC-
                            TERE SPECIAL
         JR    Z, FINI      TROUVE ?
         LD    (HL), A      DEPOSE LE CARACTERE
                            DANS LE TAMPON
         INC   HL           POINTE SUR LA CASE
                            SUIVANTE
         JR    SUIVANT      ET CONTINUE
FINI     RET
```

Exercice 8.3 : *Pour améliorer cette routine de base :*
a) Faites l'écho du caractère (s'il s'agit d'une télétype, par exemple).
b) Testez que la chaîne de caractère entrée a une longueur inférieure à 256.

Nous disposons maintenant d'une chaîne de caractères dans une zone mémoire.

Elle pourra être traitée de diverses manières.

TEST D'UN CARACTÈRE

Essayons de déterminer si le caractère situé dans la case mémoire d'adresse LOC est égal à 0,1 ou à 2 :

```
ZUD     LD    A, (LOC)      LIRE LE CARACTERE
        CP    00            EST-CE UN ZERO ?
        JP    Z, ZERO       SAUT SI OUI
        CP    01            UN ?
        JP    Z, UN
        CP    02            DEUX ?
        JP    Z, DEUX
        JP    PAS-TROUVE    SI CE N'EST AUCUN DES
                            TROIS
```

Il suffit de lire le caractère, puis d'utiliser CP pour tester sa valeur. Passons à un autre test.

TEST D'INTERVALLE

Nous déterminerons si le caractère ASCII situé à l'adresse mémoire LOC est un chiffre compris entre 0 et 9 :

```
INTER   LD    A, (LOC)      LIRE LE CARACTERE
        AND   7FH           MASQUE LE BIT DE PARITE
        CP    30H           ASCII 0
        JR    C, FINI       CARACTERE TROP PETIT ?
        CP    39H + 1       ASCII 9
        JR    NC, FINI      TROP GRAND ?
        CP    A             POSITIONNE L'INDIC. Z
FINI    RET                 SORTIR
```

Le caractère ASCII « 0 » est représenté, en hexadécimal, par « 30 » ou par « B0 », suivant qu'on recourre ou non à la parité. De manière similaire, le caractère ASCII « 9 » est représenté, en hexadécimal, par « 39 » ou « B9 ».

Le but de la seconde instruction est d'enlever le bit 7 — le bit de parité —, dans l'hypothèse où il aurait été utilisé de manière à ce que le programme

puisse s'appliquer aux deux cas. Le caractère est ensuite comparé aux caractères ASCII « 0 » et « 9 ». Avec l'instruction de comparaison, l'indicateur Z est positionné si la comparaison réussit. L'indicateur de report est positionné s'il y a report (retenue), et mis à 0 dans le cas contraire. En d'autres termes, avec l'instruction CP, l'indicateur C est positionné si la valeur du littéral qui apparaît dans l'instruction est plus grande que celle contenue dans l'accumulateur. Il sera mis à « 0 » si le littéral est plus petit ou égal.

La dernière instruction, CP A, force un « 0 » dans l'indicateur Z, dont le rôle est ici d'indiquer si le caractère contenu dans CAR est situé ou non dans l'intervalle (0,9). D'autres conventions peuvent être utilisées. Par exemple : mettre une valeur particulière dans l'accumulateur, pour indiquer le résultat du test.

Exercice 8.4 : *Le programme suivant est-il équivalent au précédent ?*

```
LD      A, (CAR)
SUB     30H
JP      M, FINI
SUB     10
JP      P, FINI
ADD     A, 10
```

Exercice 8.5 : *Déterminer si le caractère ASCII contenu dans l'accumulateur est une lettre de l'alphabet.*

En utilisant une table ASCII, vous noterez que la parité est souvent utilisée. Par exemple, le code ASCII de « 0 » est « 0110000 ». C'est un code sur 7 bits. Cependant, avec la parité impaire par exemple, il faut garantir que le nombre total de « 1 » dans le code est impair, et le code devient « 10110000 ». Un « 1 » supplémentaire a été ajouté, à gauche. Nous obtenons « B0 », en hexadécimal. Développons maintenant un programme pour générer le bit de parité.

GÉNÉRATION DE LA PARITÉ

Ce programme génère un bit de parité paire dans le bit 7 :

```
PARITE    LD     A, (CAR)     LIRE LE CARACTERE
          AND    7FH          EFFACER BIT 7
          JP     PE, FINI     TESTER SI PARITE DEJA
                              PAIRE
          OR     80H          SINON RAJOUTER LE BIT DE
                              PARITE
FINI      LD     (LOC), A     RANGER LE RESULTAT
```

Le programme utilise le circuit interne de détection de parité disponible sur le Z80.

La troisième instruction : JP PE, FINI teste si la parité du caractère dans l'accumulateur est d'ores et déjà paire. Le saut aura lieu si tel est bien le cas, autrement dit si la condition « PE » est vérifiée (« PE », de l'anglais Parity Even : parité paire).

Si la parité est impaire, le saut n'aura pas lieu. Il faut ajouter un « 1 » au code du caractère, en position 7, pour rendre sa parité paire. C'est le but de la quatrième instruction :

```
OR      80H
```

Finalement, la valeur obtenue est rangée à l'adresse LOC.

Exercice 8.6 : *La solution du problème précédent ne pose aucun problème, si on recourt au circuit interne de détection de parité du Z80. A titre d'exercice, essayez de résoudre ce même problème, sans utiliser ce circuit. Décalez le contenu de l'accumulateur, et comptez le nombre de 1 de la valeur contenue dans l'accumulateur. Vous saurez quel bit de parité il convient de mettre.*

Exercice 8.7 : *En utilisant le programme de parité ci-dessus, vérifier si la parité d'un mot est correcte. Il vous faudra, pour cela, calculer le bit de parité du mot, puis le comparer au bit de parité situé dans le mot.*

CONVERSION DE CODE ASCII EN DCB

La conversion de l'ASCII en BCD est très simple. A noter que la représentation hexadécimale des caractères ASCII de 0 à 9 est 30 à 39, ou B0 à B9, selon la parité. La représentation DCB est donc obtenue, tout simplement, en laissant tomber le « 3 » ou le « B », c'est-à-dire, en masquant le quartet de gauche.

```
ASCDCB  CALL  INTER          VERIFIER QUE LE CARAC-
                             TERE EST ENTRE 0 ET 9
        JP    NZ, ILLEGAL    SORTIR SI CE N'EST PAS LE
                             CAS
        LD    A, (CAR)       LIT LE CARACTERE
        AND   0FH            MASQUE LES 4 BITS DE
                             GAUCHE
        LD    (CARDCB), A    RANGE LE RESULTAT
```

Exercice 8.8 : *Ecrivez un programme convertissant du DCB en ASCII.*

Exercice 8.9 : *Ecrivez un programme convertissant du DCB en binaire (plus difficile).* Conseil : remarquez que N3N2N1N0 en DCB est $(((N3 \times 10) + N2) \times 10 + N1) \times 10 + N0$ en binaire.

Pour multiplier par 10, utilisez un décalage à gauche (= × 2), un second décalage à gauche (= × 4), un ADC (= × 5) et un dernier décalage (= × 10).

Dans une notation DCB complète, le premier mot doit contenir le nombre de chiffres DCB, le quartet suivant, le signe, et tous les autres, un chiffre DCB. (Nous supposerons qu'il n'y a pas de point décimal). Le dernier quartet de la chaîne peut rester inutilisé.

CONVERSION D'HEXADÉCIMAL EN ASCII

« A » contient un chiffre hexadécimal. Nous ajouterons, simplement, un « 3 » (ou un « B ») dans le quartet de gauche :

```
        AND   0FH           MET A ZERO LE QUARTET
                            DE GAUCHE
        ADD   A, 30H        ASCII
        CP    3AH           CORRECTION NECESSAIRE ?
        JP    M, FINI       SAUT SI PAS NECESSAIRE
        ADD   A, 7          SINON CORRIGER POUR LES
                            CHIFFRES DE A à F
```

Exercice 8.10 : *Convertissez de l'hexadécimal en ASCII, en supposant que vous disposez d'un format compacté (deux chiffres hexadécimaux dans A).*

TROUVER LE PLUS GRAND ÉLÉMENT D'UNE TABLE

L'adresse du début de la table se trouve à l'adresse mémoire BASE, en page zéro. Sa première entrée est le nombre d'octets qu'elle contient. Ce programme découvrira le plus grand élément de la table. Sa valeur sera laissée dans A, et son adresse dans le mot situé à l'adresse INDEX.

Il utilise les registres A, B, H et L, et l'adressage indirect, de manière à pouvoir travailler sur des tables situées à n'importe quel endroit de la mémoire. (cf. figure 8.1).

```
MAX     LD    HL, BASE      ADRESSE DE LA TABLE
        LD    B, (HL)       NOMBRE D'ELEMENTS DE
                            LA TABLE
        LD    A, 0          INITIALISER LA PLUS
                            GRANDE VALEUR
        INC   HL            HL POINTE SUR LA PRE-
                            MIERE ENTREE DE LA
                            TABLE
        LD    (INDEX), HL   INITIALISE L'INDEX
```

```
BOUCLE  CP    (HL)          COMPARE AVEC ENTREE
                            SUIVANTE
        JR    NC, NO – CH   SAUT SI INFERIEURE AU
                            MAXIMUM
        LD    A, (HL)       PRENDRE NOUVELLE
                            VALEUR MAX.
        LD    (INDEX), HL   CONSERVER POINTEUR SUR
                            NOUVEAU MAX
NO – CH INC   HL            POINTE SUR VALEUR
                            SUIVANTE
        DJNZ  BOUCLE        CONTINUER SI PAS FINI
        RET
```

Il faut d'abord tester la n-ième entrèe. Si elle est supérieure à 0, elle est mise dans A, et son adresse est conservée dans INDEX. L'entrée (n + 1) est ensuite testée, etc.

Ce programme « marche » pour les entiers positifs.

Exercice 8.11 : *Modifiez le programme pour qu'il marche aussi avec des nombres négatifs, en complément à deux.*

Exercice 8.12 : *Marcherait-il avec des caractères ASCII ?*

Exercice 8.13 : *Ecrivez un programme qui classe n nombres en ordre croissant.*

Exercice 8.14 : *Ecrivez un programme qui classe n noms de trois lettres chacun par ordre alphabétique.*

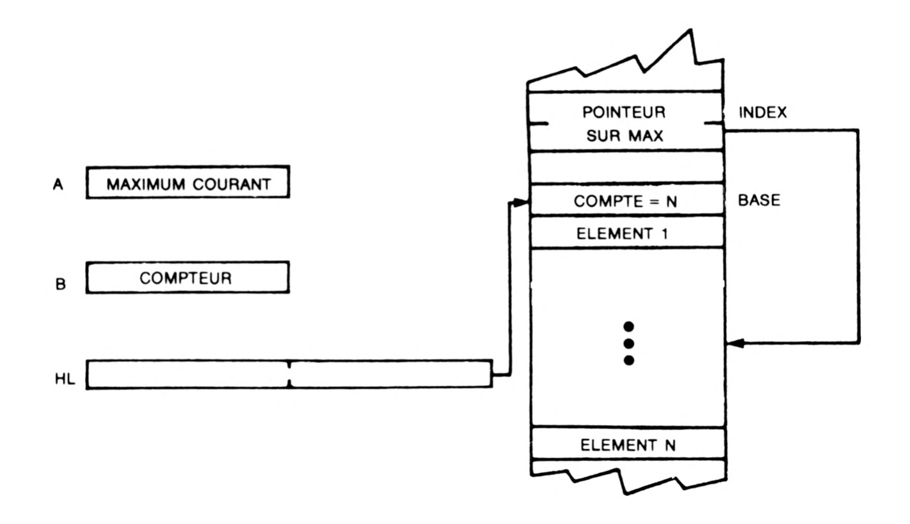


Figure 8.1. — Le plus grand élément de la table

SOMME DE N ÉLÉMENTS

Ce programme va calculer la somme, sur 16 bits, de N éléments d'une table. L'adresse du début de cette dernière se trouve à l'adresse mémoire BASE, en page zéro. Sa première entrée contient son nombre d'éléments. La somme sur 16 bits sera déposée dans les adresses mémoires SOMHAUT et SOMBAS. Si elle est trop grande pour tenir sur 16 bits, seuls les 16 bits de poids faibles seront conservés (on dit, dans ce cas, que les bits de poids forts sont tronqués).

Les registres A, B, H, L, IX sont modifiés. Le programme suppose que la table contient, au maximum, 256 éléments (voir figure 8.2).

```
SOMME     LD    HL, BASE      POINTE SUR DEBUT TABLE
          LD    B, (HL)       NOMBRE D'ELEMENTS DANS
                              LE COMPTEUR
          INC   HL            HL POINTE SUR PREMIERE
                              ENTREE
          LD    IX, SOMBAS    POINTE SUR PARTIE BASSE
                              DU RESULTAT
          LD    A, 0          INITIALISE...
          LD    (IX + 0), A   ...LA PARTIE BASSE ET..
          LD    (IX + 1), A   ..LA PARTIE HAUTE DU
                              RESULTAT
BOUCLE    LD    A, (HL)       LIT L'ENTREE
          ADD   A, (IX + 0)   CALCULE LA SOMME PAR-
                              TIELLE
          LD    (IX + 0), A   ET LA RANGE
          JR    NC,NO - RET   TESTE LA PRESENCE D'UNE
                              RETENUE
          INC   (IX + 1)      TIENT COMPTE DE LA RETE-
                              NUE SINON
NO - RET  INC   HL            POINTE SUR ENTREE
                              SUIVANTE
          DEC   B             DECREMENTE LE
                              COMPTEUR
          JR    NZ, BOUCLE    ET CONTINUER JUSQU'A LA
                              FIN
          RET
```

Il s'agit d'un programme simple, compréhensible en première lecture.

Exercice 8.15 : *Modifiez ce programme pour :*

a) calculer une somme sur 24 bits
b) calculer une somme sur 32 bits
c) détecter si un débordement a lieu

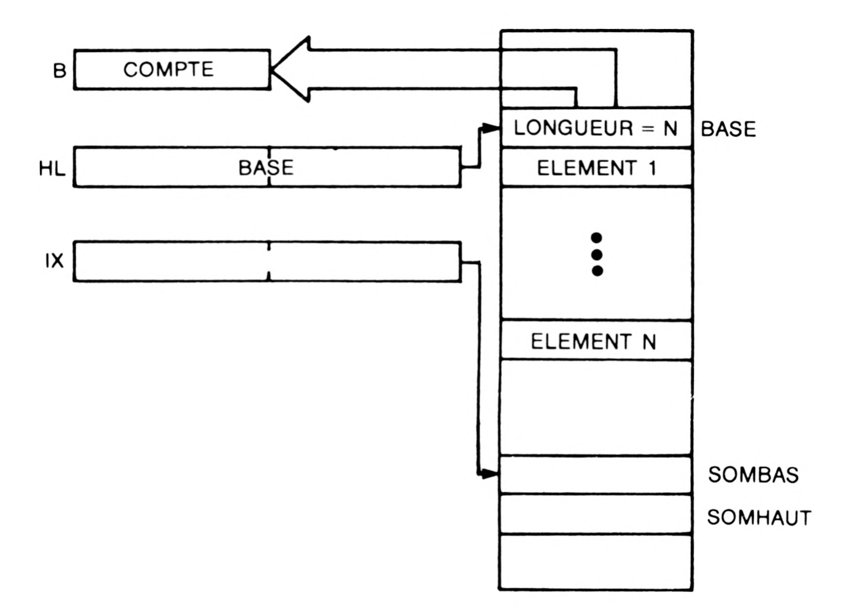


Figure 8.2. — Somme de N éléments

CALCUL D'UNE SOMME DE CONTRÔLE

Une somme de contrôle est un nombre, ou un ensemble de nombres, calculé à partir d'un bloc de caractères successifs. La somme de contrôle est calculée au fur et à mesure du rangement des données. Elle se range à la fin, derrière ces données. Pour contrôler la correction des données, il est nécessaire de les lire en recalculant la somme de contrôle, et de comparer cette somme à celle qui se trouve en mémoire. Si les deux valeurs ne sont pas égales : il y a une erreur.

Plusieurs algorithmes peuvent être utilisés. Nous effectuerons, ici, le OU exclusif de tous les octets d'une table de N éléments, et laisserons ce résultat dans l'accumulateur. Comme d'habitude, le début de la table est à l'adresse BASE, en page zéro. La première entrée de la table contient son nombre d'éléments. Le programme modifie A, B, H, L. N doit être inférieur à 256.

```
CONTROLE LD     HL, BASE      ADRESSE DE LA TABLE
                              DANS HL
         LD     B, (HL)       CHARGE N DANS LE REGIS-
                              TRE B
         XOR    A             INITIALISE LA SOMME A 0
         INC    HL            POINTE SUR LA PREMIERE
                              ENTREE
```

```
BOUCLE   XOR   (HL)           CALCULE LA SOMME
         INC   HL             POINTE SUR L'ELEMENT
                              SUIVANT
         DEC   B              DECREMENTE LE
                              COMPTEUR
         JR    NZ, BOUCLE     BOUCLER SI PAS FINI
         LD    (SOMME), A     RANGER LA SOMME A LA
                              FIN

         RET
```

COMPTER DES ZÉROS

Ce programme compte le nombre de zéros dans notre table habituelle, et met le résultat à l'adresse TOTAL. Il modifie A B C H L.

```
ZEROS    LD    HL, BASE       POINTE SUR LA TABLE
         LD    B, (HL)        LONGUEUR TABLE DANS B
         LD    C, 0           INITIALISE LE TOTAL
         INC   HL             POINTE SUR PREMIERE
                              ENTREE
ZBOUCLE  LD    A, (HL)        CHARGER L'ELEMENT DANS
                              A
         OR    0              POSITIONNE L'INDICATEUR
                              Z
         JR    NZ, PASZERO    SAUT SI DIFFERENT DE
                              ZERO
         INC   C              SINON, AJOUTER 1 AU
                              TOTAL
PASZERO  INC   HL             POINTE SUR ENTREE
                              SUIVANTE
         DEC   B              DECREMENTE LE
                              COMPTEUR
         JR    NZ, BOUCLE
         LD    A, C
         LD    (TOTAL), A     SAUVER LE RESULTAT
```

Exercice 8.16 : *Modifiez ce programme pour compter :*
*a) le nombre d'étoiles de la table (le caractère « * ») ;*
b) le nombre de lettres de l'alphabet ;
c) le nombre de chiffres compris entre « 0 » et « 9 ».

TRANSFERT DE BLOC

Prenons dans un bloc commençant à l'adresse SOURCE, une entrée sur trois et mettons-la dans un bloc commençant à l'adresse DEST.

```
FER3      LD     HL, SOURCE
          LD     DE, DEST      INITIALISE LES POINTEURS
          LD     BC, TAILLE
BOUCLE    LDI                  TRANSFERT AUTOMATIQUE
          INC    HL
          INC    HL
          JP     PE, BOUCLE
```

TRANSFERT D'UN BLOC DCB

Nous allons remonter d'un « cran » en mémoire une chaîne de chiffres en DCB compacté. Nous allons, autrement dit, décaler des quartets (voir figure 8.3).

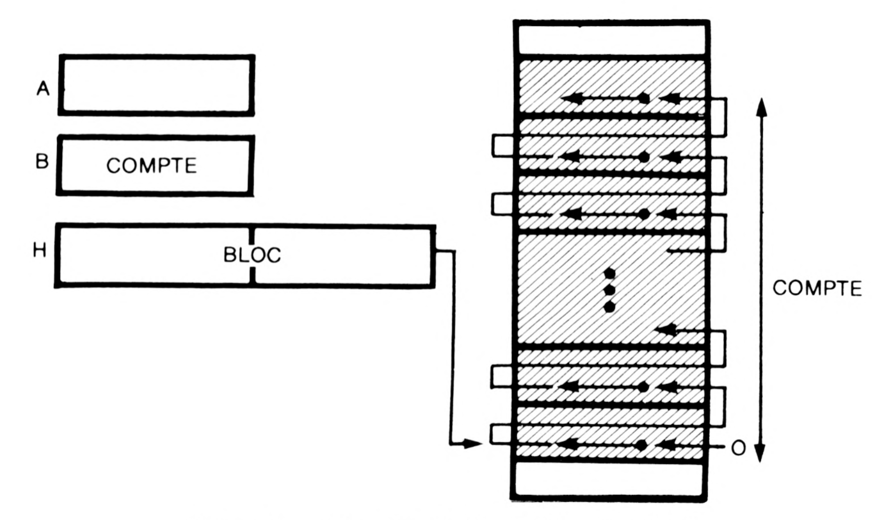


Figure 8.3. — Transfert de bloc DCB — la mémoire

Voici le programme :

```
REMONTEE LD     B, COMPTE
         LD     HL, BLOC
         XOR    A              A = 0
BOUCLE   RLD
         DEC    HL             POINTE SUR ELEMENT
                               SUIVANT
         DJNZ   BOUCLE
```

Le programme fait appel à l'instruction RLD, que nous n'avons pas encore utilisée.

RLD effectue une rotation de chiffres DCB entre A et (HL). (HL) et M désignent le contenu de l'adresse mémoire pointée par HL.

M PARTIE BASSE va dans M PARTIE HAUTE
M PARTIE HAUTE va dans A PARTIE BASSE
A PARTIE BASSE va dans M PARTIE BASSE

Ici, « basse » et « haute » désignent les deux quartets contenus dans un octet.

Pour utiliser la puissante instruction DJNZ, nous avons pris le registre B comme compteur. HL est initialisé pour pointer au début du bloc.

A est utilisé pour conserver le chiffre DCB de gauche, qui est déplacé à chaque rotation, entre deux accès successifs au bloc.

Par convention, nous mettrons un « 0 » à la fin du bloc.

COMPARAISON DE DEUX NOMBRES SIGNÉS SUR 16 BITS

IX pointe sur le premier nombre N1.
IY pointe sur N2 (voir figure 8.4).

Le programme met l'indicateur C à 1 si N1 < N2, et l'indicateur Z à 1 si N1 = N2.

```
COMPARE  LD    B, (IX + 1)     CHERCHER SIGNE DE N1
         LD    A, B
         AND   80H             TESTE LE SIGNE, MET INDI-
                               CATEUR C A ZERO
         JR    NZ, NEGM1       SAUT SI N1 NEGATIF
         BIT   7, (IY + 1)
         RET   NZ              N2 EST NEGATIF
         LD    A, B
         CP    (IY + 1)        LES SIGNES SONT TOUS LES 2
                               POSITIFS
         RET   NZ
         LD    A, (IX)
         CP    (IY)
         RET
NEGM1    XOR   (IY + 1)
         RLA                   BIT DE SIGNE DANS INDICA-
                               TEUR C
         RET   C               LES SIGNES SONT DIFFE-
                               RENTS
         CP    (IY + 1)        LES 2 SIGNES SONT
                               NEGATIFS
         RET   NZ
         LD    A, (IX)
         CP    (IY)
         RET
```

Le programme commence par tester les signes de N1 et de N2. Si N1 est négatif, un saut se produit vers NEGH1. Sinon, la première moitié du programme est exécutée.

A noter que l'instruction BIT est utilisée à la cinquième ligne, pour tester le signe de N2 directement en mémoire :

```
        BIT     7, (IY + 1)
```

Nous aurions pu faire la même chose pour N1, à ceci près que nous aurons, très vite, besoin de sa valeur. C'est pourquoi il est plus simple de lire N1 en mémoire, et de le préserver dans B !

```
COMPARE LD      B, (IX + 1)
```

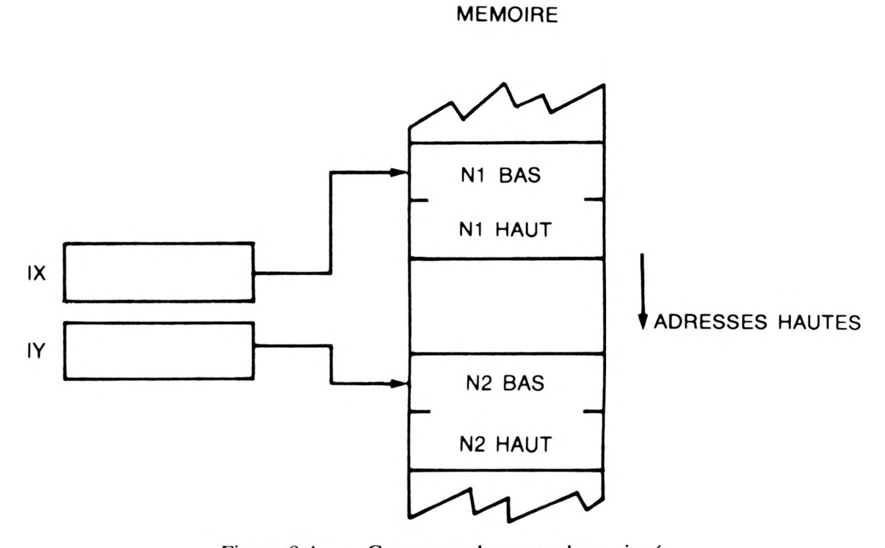


Figure 8.4. — Comparer deux nombres signés

Il est nécessaire de préserver N1 dans B, car l'instruction AND peut détruire le contenu de A :

```
        LD      A, B
        AND     80H
```

Remarquez l'utilisation de l'instruction de retour conditionnel (ligne 6) :

```
        RET     NZ
```

C'est là une puissante propriété du Z80, qui simplifie grandement la programmation.

L'instruction de comparaison travaille directement sur le contenu de la mémoire, en mode indexé.

```
CP      (IY + 1)
```

Lorsqu'on compare deux nombres, il faut comparer les deux octets de poids forts, avant ceux de poids faible.

Notez, dans ce programme, l'utilisation intensive du mécanisme d'indexation, qui procure un code efficient.

TRI PAR BULLES

Il s'agit d'une technique de tri utilisée pour classer les éléments d'une table par ordre croissant ou décroissant. Son nom lui vient d'une analogie. Le plus petit élément remonte, en effet, doucement jusqu'au début de la table, comme une bulle à la surface d'un liquide. Chaque fois qu'il rencontre un élément « plus lourd », il le franchit d'un saut.

Un exemple pratique du tri par bulles est présenté à la figure 8.5. La liste à trier contient : (10, 5, 0, 2, 100) et doit l'être par ordre décroissant (« 0 » en haut). L'algorithme est simple, et son ordinogramme présenté à la figure 8.7.

Comparons les deux éléments du haut (ou ceux du bas). Si le plus bas des deux est aussi le plus petit (le plus « léger »), ils devront être échangés. Sinon, rien. Pour des raisons pratiques, l'échange éventuel sera noté dans un indicateur que nous appellerons « ÉCHANGES ». Le processus est, ensuite, répété sur la paire suivante d'éléments, jusqu'à ce que tous les éléments aient été comparés deux à deux.

La première passe est illustrée par les étapes 1 2 3 4 5 6 de la figure 8.5, en allant du bas vers le haut. (L'inverse était tout aussi justifié.)

Si, au cours d'une passe, aucun échange n'a lieu, le tri est terminé. Si un seul se produit, une autre passe est nécessaire.

Dans notre exemple, quatre passes sont nécessaires (figure 8.6).

Le processus est simple, et largement utilisé.

Une complication supplémentaire provient du mécanisme pratique de l'échange.

Pour échanger A et B, il est, en effet, impossible d'écrire :

```
A = B
B = A
```

Nous perdrions ainsi la valeur de A (essayez sur un exemple).

La solution correcte consiste à utiliser une variation auxiliaire pour préserver cette valeur :

```
TEMP = A
A    = B
B    = TEMP
```

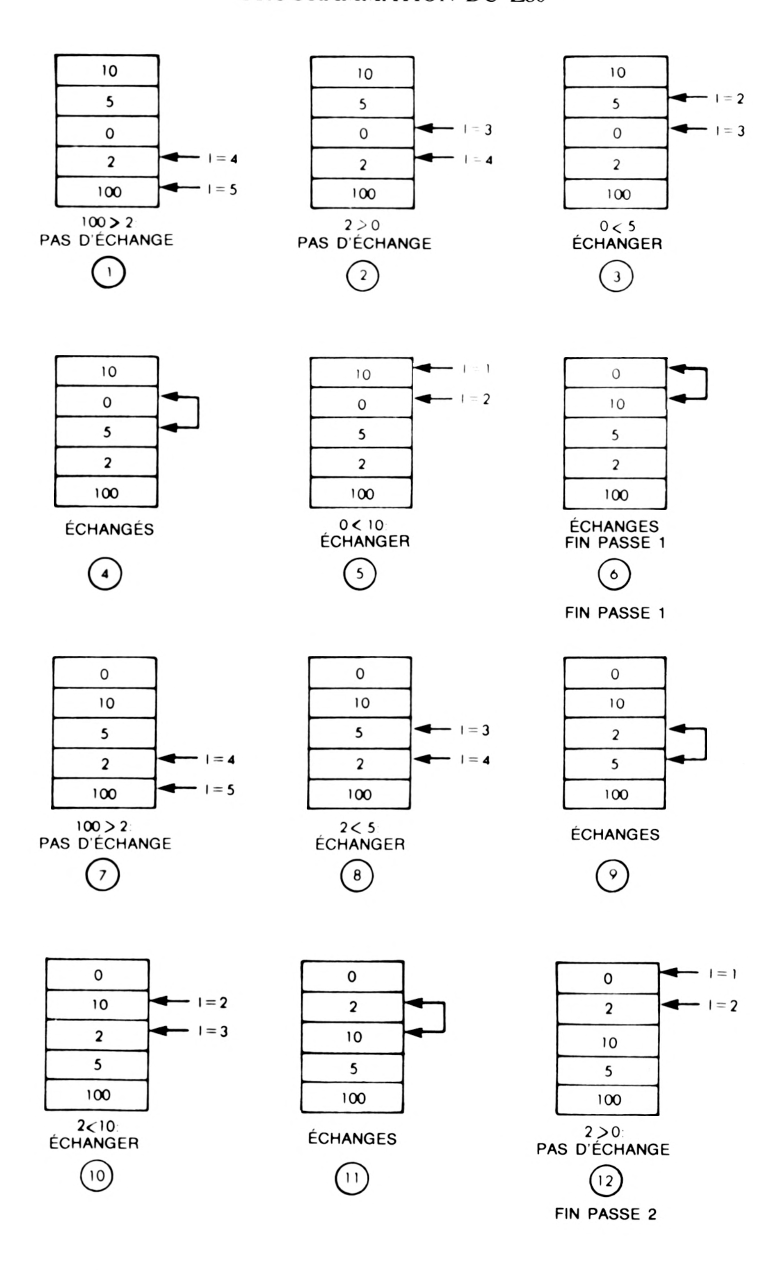


Figure 8.5. — Exemple de tri par bulles : phases 1 à 12

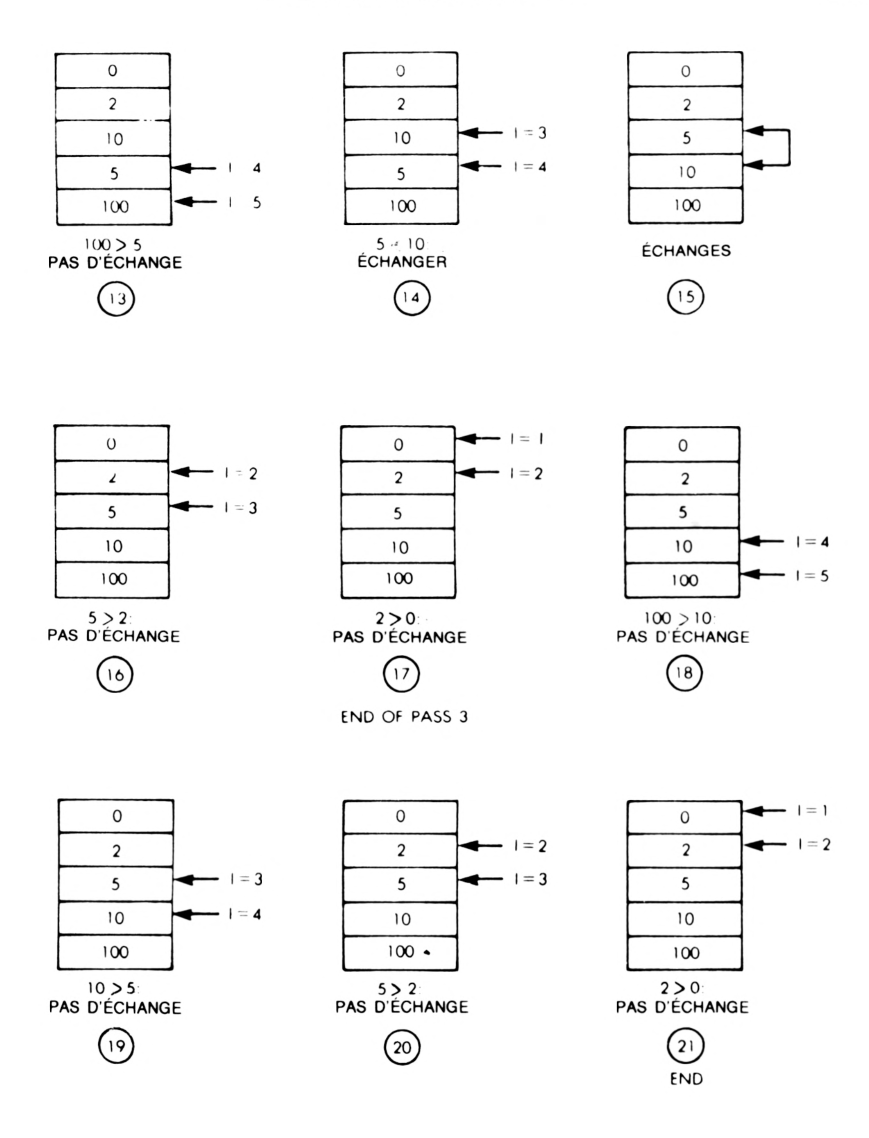


Figure 8.6. — Exemple de tri par bulles : phases 13 à 21

Ça marche (essayez sur un exemple). C'est ce qu'on appelle une permutation circulaire.

Tous les programmes procèdent de cette manière pour opérer un échange (voir l'ordinogramme de la figure 8.7) :

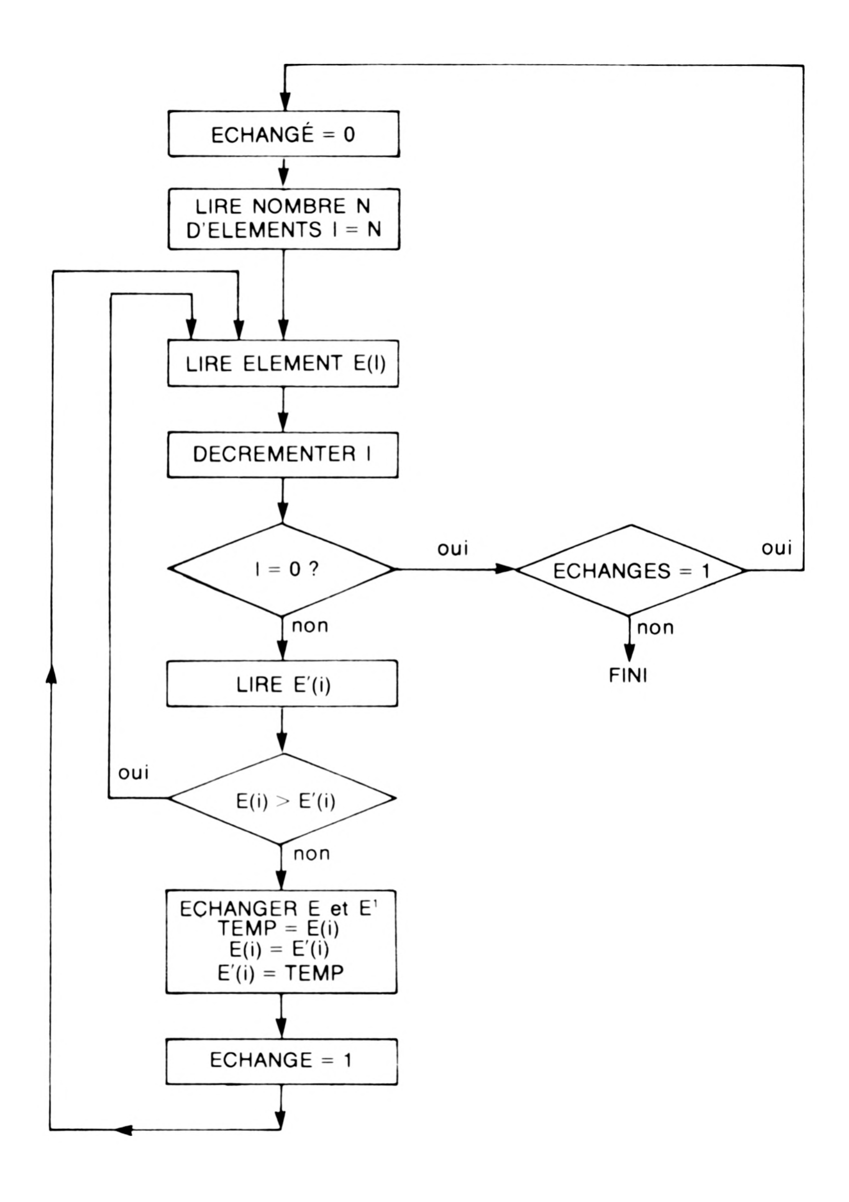


Figure 8.7. — Ordinogramme du tri par bulles

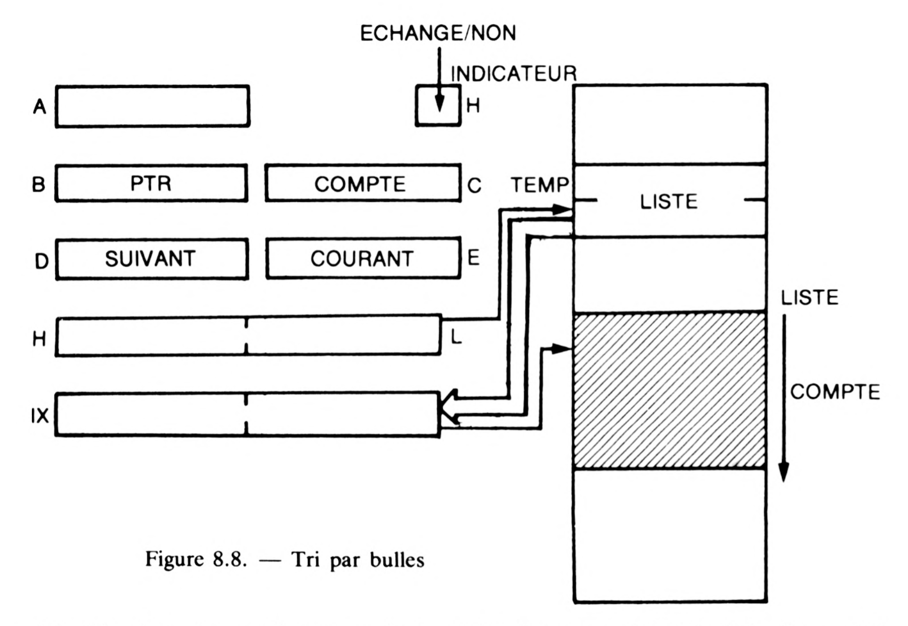


Figure 8.8. — Tri par bulles

L'utilisation des registres et de la mémoire est présentée à la figure 8.8. Le programme est le suivant :

TRI	LD	(TEMP), HL	SAUVEGARDE DE HL
ENCORE	LD	IX, (TEMP)	(IX) = (HL)
	RES	ECHANGES, H	INDIQUER = PAS D'ECHANGE
	LD	B, C	
	DEC	B	
SUIVANT	LD	A, (IX)	
	LD	D, A	ENTREE COURANTE
	LD	E, (IX + 1)	ENTREE SUIVANTE
	CP	E	COMPARAISON
	JR	NC, NO − ECH	SAUT SI COURANT SUPERIEUR A SUIVANT
ECHANGER	LD	(IX), E	METTRE SUIVANT A LA PLACE DE COURANT
	LD	(IX + 1), D	ET VICE VERSA
	SET	ECHANGES, H	INDIQUER QU'ON A FAIT UN ECHANGE
NO − ECH	INC	IX	PASSER A LA PAIRE SUIVANTE
	DJNZ	SUIVANT	CONTINUER LA PASSE
	BIT	ECHANGES, H	TESTER SI ECHANGES EN FIN DE PASSE
	JR	NZ, ENCORE	NOUVELLE PASSE SI NECESSAIRE
	RET		

CONCLUSION

Nous venons de présenter des routines d'utilité générale, qui utilisent des combinaisons de techniques décrites dans les précédents chapitres. Ces exemples devraient vous permettre de commencer à écrire vos propres programmes. Un grand nombre de ces routines utilise une structure de données particulière : la table.

Il existe d'autres modalités de structuration des données. Nous allons maintenant les passer en revue.

9

STRUCTURES DE DONNÉES

1ère PARTIE — THÉORIE

INTRODUCTION

L'obtention d'un bon programme est soumise à deux opérations de conception : celle de *l'algorithme* et celle des structures de données. La plupart des programmes simples ne comportent pas de structures de données particulières, de sorte que le principal objectif, lors de l'apprentissage de la programmation, est de concevoir des algorithmes, et de les coder efficacement, dans un langage machine donné. C'est ce que nous avons fait jusqu'ici. Cependant, la conception de programmes plus complexes implique, en outre, de comprendre les structures de données. Deux d'entre elles ont déjà été utilisées au cours de ce livre : la table et la pile. Le but de ce chapitre est d'en présenter d'autres, plus générales, que vous pourrez être amenés à utiliser. Ce chapitre est totalement indépendant du microprocesseur, ou même de l'ordinateur retenu. Il ne fait appel qu'à la théorie, et englobe l'organisation logique des données dans le système. Il existe, sur ce sujet, des livres spécialisés, de même qu'il en existe pour la multiplication, la division, et les autres algorithmes usuels. Ce chapitre se limitera donc aux seules structures essentielles. Il ne prétend pas être exhaustif.

Passons maintenant en revue les structures de données les plus courantes.

LES POINTEURS

Un pointeur est un nombre qui désigne l'emplacement de la donnée proprement dite. Tout pointeur est une adresse. Mais inversement, chaque adresse ne mérite pas nécessairement le nom de pointeur. Une adresse ne peut être assimilée à un pointeur, que si elle pointe sur quelque chose qui

constitue une donnée, ou sur une information structurée. Nous avons déjà rencontré un exemple typique de pointeur : le pointeur de pile, qui pointe sur le sommet de la pile (ou souvent juste au-dessus). Nous verrons que la pile est une structure de donnée usuelle, appelée LIFO.

Autre exemple, lors de l'utilisation de l'adressage indirect : l'adresse d'indirection pointe toujours sur la donnée à laquelle on veut accéder.

Exercice 9.1 : *Soit la figure 9.1. A l'adresse 15 de la mémoire, se trouve un pointeur sur la table T. T commence à l'adresse 500. Quel est le contenu effectif du pointeur sur T ?*

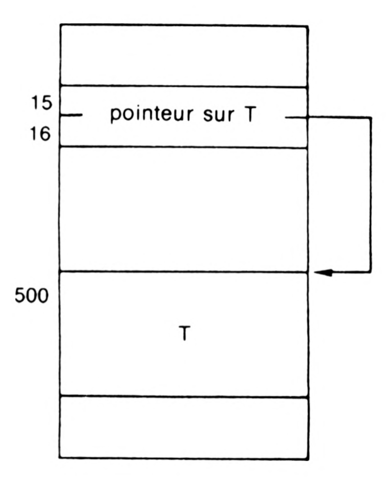


Figure 9.1. — Un pointeur d'indirection

LES LISTES

La presque totalité des structures de données sont organisées en listes de toutes natures.

Les listes séquentielles

La liste séquentielle, ou table, ou encore bloc, est probablement la structure de données la plus simple. Nous l'avons, d'ailleurs, déjà utilisée. Les tables sont normalement ordonnées selon un critère donné. Par exemple, l'ordre alphabétique ou numérique. Il est ainsi facile de retrouver un élément dans la table au moyen, notamment, de l'adressage indexé. Un bloc désigne habituellement un groupe de données possédant des limites bien définies, mais dont le contenu n'est pas ordonné. Ce dernier peut

contenir une chaîne de caractères ; il peut s'agir d'un secteur sur un disque, ou de quelque zone logique (appelée segment) de la mémoire.

Pour retrouver facilement les blocs d'information, il faut utiliser les catalogues ou répertoires.

Les répertoires

Un répertoire est une liste de tables ou de blocs. Un système de fichiers utilise normalement une structure de répertoire. A titre d'exemple simple, précisons que le répertoire principal peut contenir une liste de noms d'utilisateurs [cf. figure 9.2]. L'entrée associée à l'utilisateur « Martin » pointe sur le fichier répertoire de Martin. Ce dernier est une table contenant les noms de tous les fichiers de Martin, et leur emplacement. C'est encore une table de pointeurs. Dans ce cas, nous nous sommes contentés de concevoir un répertoire à deux niveaux. Un système souple autoriserait l'insertion de répertoires intermédiaires supplémentaires, au gré de l'utilisateur.

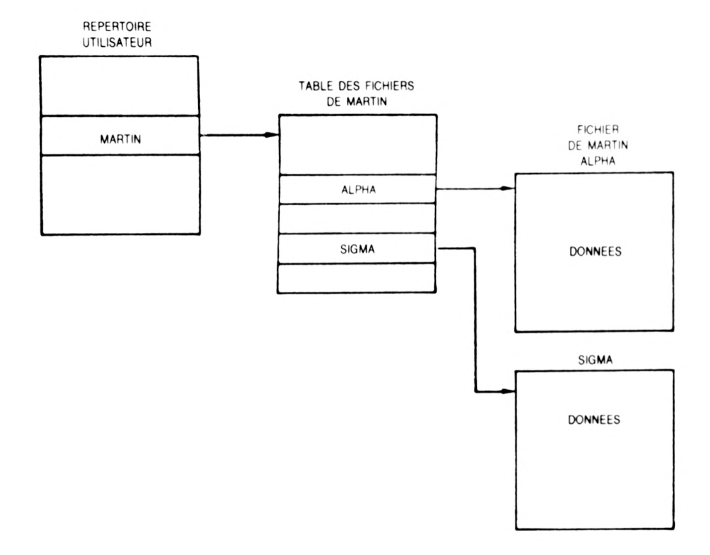


Figure 9.2. — Une structure de répertoire

La liste chaînée

Dans un système, des blocs d'information représentent souvent des données, des événements, ou d'autres structures, qui ne peuvent être déplacées rapidement. S'ils pouvaient l'être, nous les rassemblerions

probablement dans une table, de façon à les classer, ou à les structurer. Le problème, en fait, est que nous désirons les laisser là où ils sont, tout en les classant par ordre chronologique : premier, second, troisième, etc... Nous utiliserons pour résoudre ce problème une liste chaînée. Ce concept est illustré par la figure 9.3, où nous apercevons qu'un pointeur de liste, appelé PREMIERBLOC, pointe sur le début du premier bloc. Un emplacement réservé dans le bloc1, par exemple le premier ou le dernier, contient un pointeur, appelé PTR1, sur bloc2. Le procédé se répète pour bloc2 et pour bloc3. Bloc3 étant le dernier élément de la liste, son pointeur PTR3, par convention, soit contiendra une valeur spéciale « nil », soit pointera sur lui-même, de façon que la fin de la liste puisse être détectée. Cette structure est économique, puisqu'elle ne nécessite que quelques pointeurs (un par bloc), et libère l'utilisateur du souci de déplacer physiquement les blocs en mémoire.

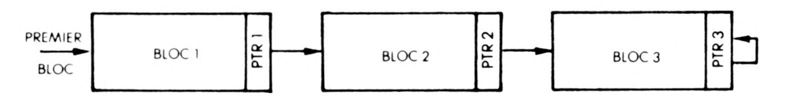


Figure 9.3. — Une liste chaînée

Examinons, par exemple, la manière dont un nouveau bloc peut être inséré [figure 9.4]. Supposons que le nouveau bloc soit à l'adresse NOUVEAUBLOC, et qu'il faille l'insérer entre bloc1 et bloc2. Il suffira, simplement, de changer la valeur de PTR1 en NOUVEAUBLOC, de sorte qu'il pointe sur blocX. PTRX contiendra, lui, l'ancienne valeur de PTR1 ; c'est-à-dire pointera sur bloc2. Les autres pointeurs de la structure restent inchangés. Nous voyons donc que l'insertion d'un nouveau bloc exige simplement la mise à jour de deux pointeurs, dans la structure. L'efficacité est évidente.

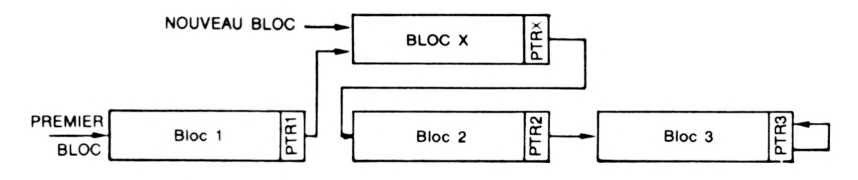


Figure 9.4. — Insertion d'un nouveau bloc

Exercice 9.2 : *Dessinez un diagramme exposant la manière dont bloc2 peut être enlevé de cette structure.*

Plusieurs types de listes ont été développés en vue de faciliter les méthodes spécifiques d'accès, d'insertion et de suppression d'un élément. Examinons quelques types de listes chaînées, parmi les plus fréquemment utilisés.

La file

La file est aussi appelée structure FIFO (de l'anglais « first in-first out » : premier entré premier sorti) [cf. figure 9.5]. Pour clarifier ce schéma, supposons, par exemple, que le bloc représenté à gauche soit un programme de service, destiné à un organe de sortie tel qu'une imprimante. Les blocs représentés à droite seront les blocs de requêtes d'impression émanant de différents programmes. L'ordre de traitement sera établi par la file d'attente. Nous voyons que le premier événement traité sera bloc1, le suivant bloc2, puis bloc3. Dans une file, la convention est que tout événement nouveau arrivant dans la file soit inséré à la fin. Dans notre exemple, il serait inséré derrière PTR3. C'est la garantie que le premier bloc inséré dans la file sera bien le premier traité. Il est assez courant, dans un système informatique, d'avoir des files pour un certain nombre d'événements devant attendre une ressource critique, telle que le processeur ou un organe d'entrée-sortie.

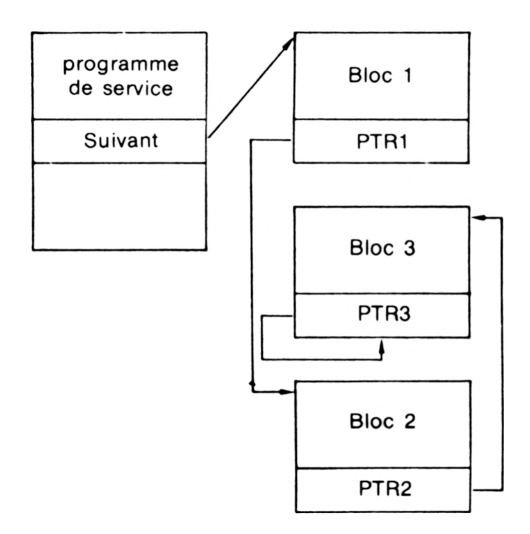


Figure 9.5. — Une file

La pile

Nous avons déjà étudié, en détail, la structure de pile. Il s'agit d'une structure LIFO (de l'anglais « last in first out » : dernier entré premier sorti). Le dernier élément déposé sur la pile est le premier à y être repris. Une pile peut être implantée soit sous forme d'un bloc ordonné, soit sous forme d'une liste. Dans la mesure où, dans les microprocesseurs, la plupart

des piles sont utilisées pour des événements rapides (appels de sous-programmes ou interruptions, par exemple), un bloc contigu sera plus volontiers alloué à la pile qu'une liste chaînée.

Liste chaînée et bloc

De la même façon, la file peut être implantée sous forme d'un bloc d'emplacements réservés. L'avantage du recours à un bloc contigu réside dans la rapidité de recherche et l'élimination des pointeurs. L'inconvénient, c'est qu'il faut, habituellement, allouer un bloc de fort grandes dimensions, pour tenir compte de la taille la plus défavorable que puisse atteindre la structure. De plus, l'insertion ou la suppression d'éléments du bloc est difficile, voire incommode. Dans la mesure où la mémoire est traditionnellement une ressource rare, il est préférable, en pratique, de réserver les blocs pour les structures de taille fixe, ou exigeant une très grande vitesse de mise à jour, à l'instar de la pile.

Liste circulaire

Une liste circulaire [ou anneau] est une liste chaînée dans laquelle la dernière entrée pointe à nouveau sur le premier bloc (cf. figure 9.6) Dans ce cas, un pointeur est souvent gardé sur le *bloc courant.* Si des événements, ou des programmes, attendent d'être servis, le pointeur *d'événement courant* sera déplacé, à chaque fois, d'une position vers la gauche ou vers la droite. Une liste circulaire correspond, en règle générale, à une structure dans laquelle tous les blocs sont supposés avoir la même priorité. Cependant, elle peut aussi être utilisée comme sous-cas d'autres structures. Tout simplement pour faciliter, lors d'une recherche, le retour au premier bloc, après l'accès au dernier.

Exemple de liste circulaire : un programme de scrutation progresse généralement le long d'un anneau, interrogeant tous les périphériques pour revenir ensuite au premier.

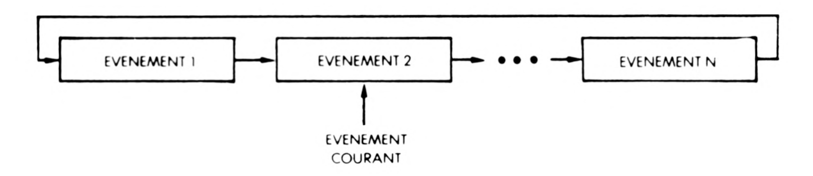


Figure 9.6. — Liste circulaire

Les arbres

Chaque fois qu'il existe une relation logique entre tous les éléments d'une structure (ce qu'on appelle généralement une syntaxe), il est possible d'utiliser une structure d'arbre. Structure, dont l'exemple le plus simple est, peut-être, l'arbre généalogique. A la figure 9.7, nous voyons que Simon a eu deux enfants : un fils, Robert et une fille, Jeanne. Jeanne, à son tour, a eu trois enfants : Elizabeth, Thomas et Philippe. Puis Thomas, deux autres enfants : Marcel et Christian. Par contre, Robert, sur la gauche du dessin, n'a pas eu de descendants.

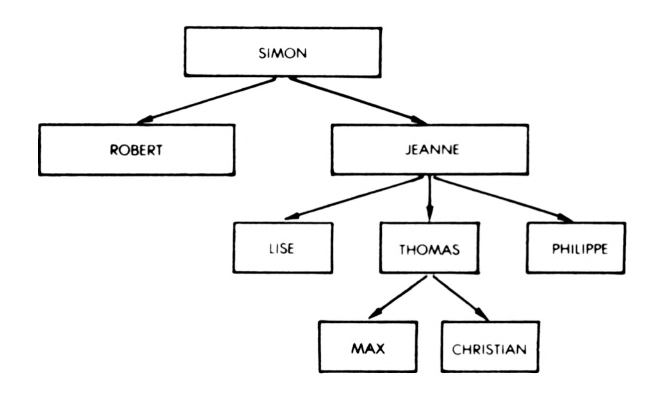


Figure 9.7. — Un arbre généalogique

Il s'agit d'un arbre structuré. Nous avons, en fait, déjà rencontré un exemple d'arbre simple, à la figure 9.2. La structure de répertoire est un arbre à deux niveaux. Les arbres sont avantageusement utilisés chaque fois que des éléments doivent être classés selon une structure fixe. Cela facilite insertion et recherche. De plus, les arbres peuvent constituer des groupes d'information, sous une forme structurée qui peut être nécessaire au traitement ultérieur. C'est le cas, notamment, dans la conception d'un compilateur ou d'un interprète.

Les listes doublement chaînées

Des liens supplémentaires peuvent être établis entre les éléments d'une liste. L'exemple le plus simple en est la liste doublement chaînée, telle qu'elle apparaît à la figure 9.8. Nous y voyons la suite habituelle de liens, orientée de gauche à droite, mais doublée, cette fois, d'une autre suite, orientée de droite à gauche. Le but de l'opération est d'accéder à l'élément qui précède immédiatement l'élément en cours de traitement, aussi facilement qu'à celui qui le suit.

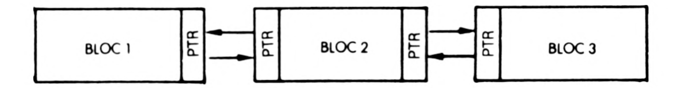


Figure 9.8. — Liste doublement chaînée

RECHERCHE ET TRI

La recherche et le tri d'éléments d'une liste dépendent directement du type de structure utilisée. De nombreux algorithmes de tri ont été développés pour les structures de données les plus couramment utilisées. Pour notre part, nous avons déjà eu recours à l'adressage indexé, qui est possible chaque fois que les éléments d'une table sont ordonnés selon un critère connu. Ces éléments peuvent être retrouvés par leurs numéros.

La *recherche séquentielle* désigne l'examen, linéaire et progressif, d'un bloc entier. La méthode est, de toute évidence, inefficace, mais il peut être utile d'y recourir, faute de mieux, les éléments n'étant pas ordonnés.

La *recherche dichotomique,* ou *logarithmique,* s'efforce de trouver un élément dans une liste triée, en divisant, à chaque étape, l'intervalle de recherche de moitié. Supposons que notre recherche se fasse dans une liste alphabétique. Nous commencerons, par exemple, au milieu de la table, et déterminerons si le nom recherché se trouve en amont ou en aval. S'il est en aval, nous éliminerons la première moitié de la table, et considérerons l'élément du milieu de la seconde moitié. De nouveau, nous comparerons cet élément à l'objet de notre recherche, en nous restreignant à l'une des moitiés de la table. Et ainsi de suite. La longueur maximum de la recherche est alors, nécessairement, $\log_2 n$, où n est le nombre d'éléments de la table.

Il existe bien d'autres techniques de recherche.

RÉCAPITULATIF

Cette section n'avait d'autre but qu'une rapide présentation des structures de données courantes, que le programmeur est susceptible d'utiliser. Quoique la plupart de ces structures aient été groupées en types, et aient reçu un nom, l'organisation globale des données dans un système complexe peut être soumise à n'importe quelle combinaison, ou peut nécessiter du programmeur qu'il invente des structures plus appropriées. Le champ des possibilités n'est de ce point de vue, limité que par l'imagination de ce dernier. De même, un grand nombre de techniques de tri et de recherche ont été développées en liaison avec les structures de données habituelles. Un exposé complet déborderait du cadre de ce livre. Cette section avait, en fait, pour objectif de mettre en relief l'importance de la conception de structures appropriées aux données à manipuler, et de fournir, à cet effet, les outils appropriés.

2ᵉ PARTIE — EXEMPLES DE CONCEPTION

INTRODUCTION

Nous allons maintenant présenter des exemples réels de conception, concernant des structures de données typiques : table, liste ordonnée, liste chaînée. Pour chacune d'entre elles, nous programmerons des algorithmes de recherche, d'insertion et de suppression.

Le lecteur intéressé par ces techniques de programmation avancée est encouragé à analyser, en détail, les programmes présentés.

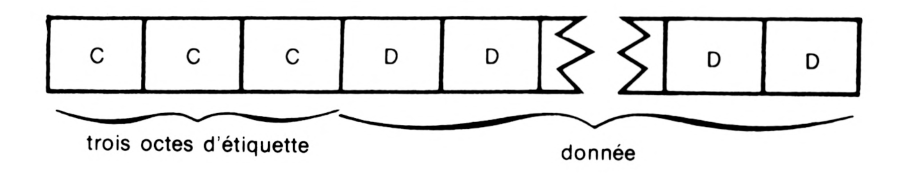
Le programmeur débutant pourra, dans un premier temps, s'épargner la lecture de cette section. Il y reviendra lorsqu'il s'y sentira prêt.

Une bonne compréhension des concepts présentés dans la première partie de ce chapitre est indispensable pour suivre ces exemples de conception. De plus, les programmes utiliseront tous les modes d'adressage du Z80, et intégreront nombre de concepts et de techniques présentés dans les chapitres précédents.

Introduisons, maintenant, trois structures : une liste simple, une liste alphabétique et une liste chaînée avec répertoire. Pour chacune d'entre elles, nous développerons trois programmes : recherche, insertion et suppression.

REPRÉSENTATION DES DONNÉES DE LA LISTE :

La liste simple et la liste alphabétique utiliseront une représentation commune de chaque élément de la liste :



Chaque élément, ou « entrée », comprend une étiquette de trois octets, et un bloc de données de n octets, n étant compris entre 1 et 253. Ainsi, chaque entrée utilise, au plus, une page (256 octets). A l'intérieur d'une même liste, tous les éléments ont la même longueur (voir figure 9.10).

Les programmes utiliseront des conventions communes de variables :

ENTLEN est la longueur d'un élément. Si chaque élément a, par exemple, 10 octets de donnée, ENTLEN = 3 + 10 = 13.

TABASE est le début de la liste, ou de la table, en mémoire.
POINTR est un pointeur sur l'élément courant.
OBJECT est l'entrée courante à localiser, à insérer ou à détruire.
TABLEN est le nombre d'entrées.

Toutes ces étiquettes sont supposées distinctes. Changer ces conventions nécessiterait des modifications mineures aux programmes.

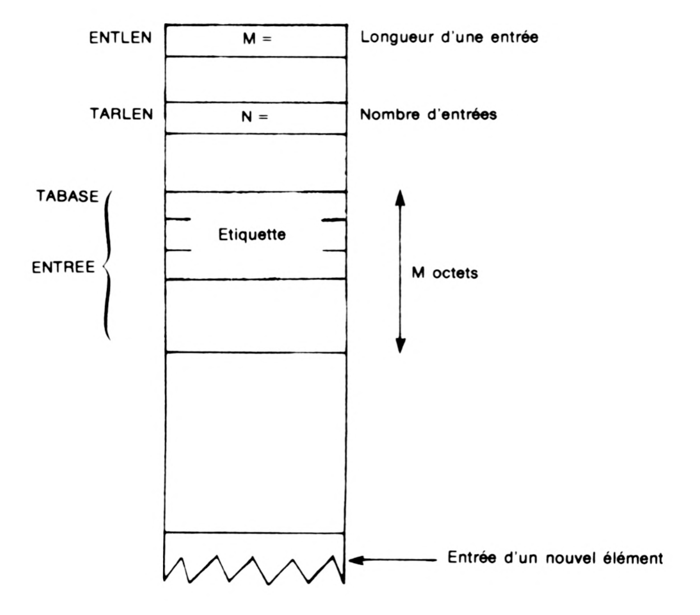


Figure 9.9. — La structure de table

UNE LISTE SIMPLE

La liste simple est organisée en une table de n éléments non triés (voir figure 9.11). Lors d'une recherche, il faut examiner une à une toutes les entrées de la liste, jusqu'à atteindre l'entrée cherchée, ou la fin de la table. Lors d'une insertion, les nouvelles entrées sont ajoutées derrière les anciennes. Lorsqu'une entrée est effacée, les entrées situées, éventuellement, plus haut en mémoire, seront décalées vers le bas, pour maintenir la table contiguë.

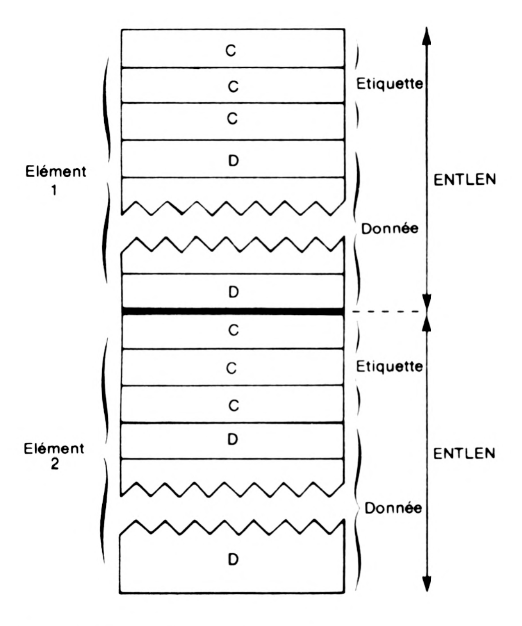


Figure 9.10. — Entrées d'une liste en mémoire

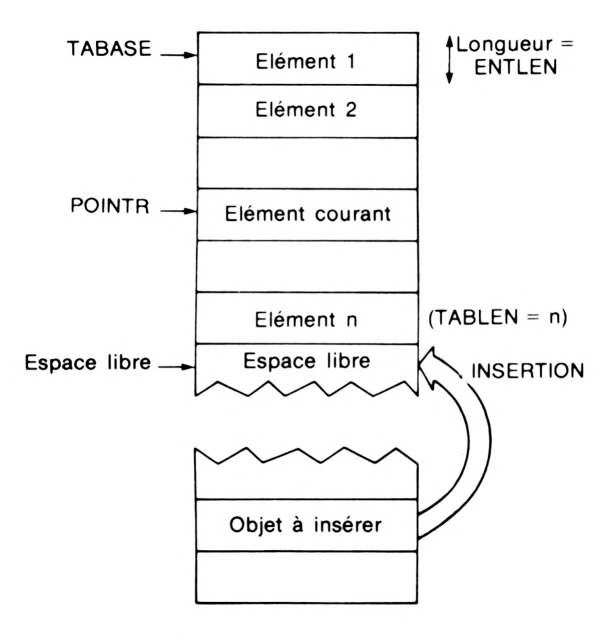


Figure 9.11. — La liste simple

Recherche

Nous utiliserons une recherche série. Le champ étiquette de chaque entrée est comparé, lettre à lettre, à l'étiquette de OBJET.

Le pointeur courant POINTR est initialisé par la valeur de TABASE.

La recherche a lieu de manière évidente. L'ordinogramme correspondant est représenté à la figure 9.12. La figure 9.16, à la fin de ce paragraphe, donne le programme correspondant (programme « SEARCH »). Un exemple d'exécution du programme est inclus à la figure 9.17.

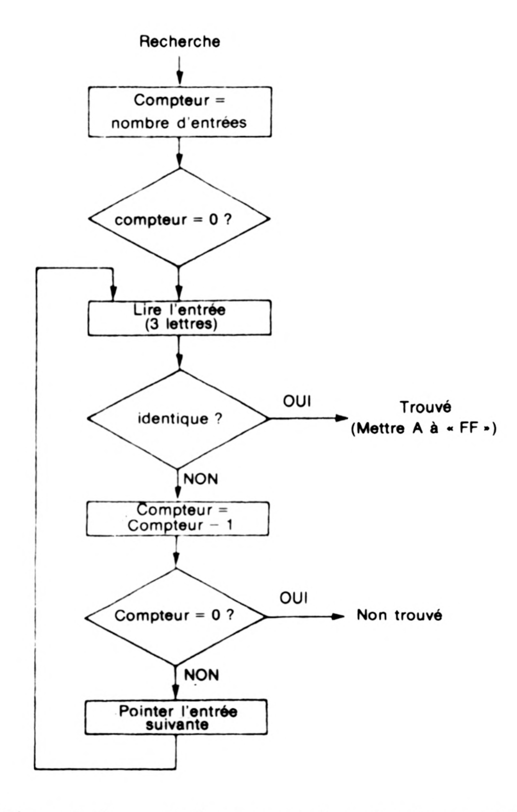


Figure 9.12. — Ordinogramme de recherche en table

Insertion

Lors de l'insertion d'un nouvel élément, le premier bloc disponible de ENTLEN octets, à la fin de la liste, est utilisé (voir figure 9.11).

Le programme vérifie d'abord que la nouvelle entrée ne figure pas déjà dans la liste (nous supposerons, ici, que toutes les étiquettes sont distinctes).

Si elle n'y figure pas, le programme incrémentera la longueur de la liste TABLEN, et placera la nouvelle entrée OBJECT à la fin de la liste. La figure 9.13 donne l'ordinogramme correspondant.

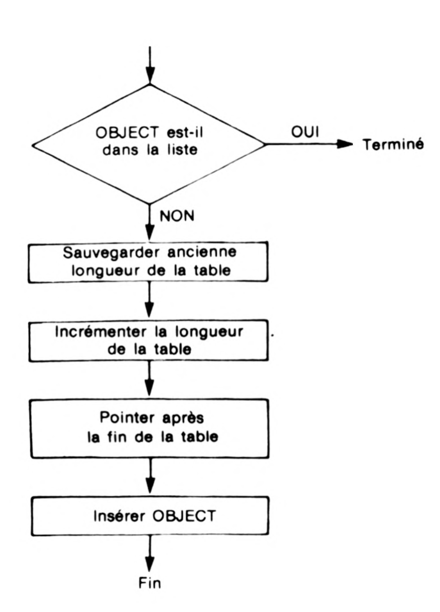


Figure 9.13. — Ordinogramme d'insertion dans la table

Le programme (figure 9.16) est appelé « NEW ». Il occupe les emplacements mémoire de 0135 à 015E.

Le registre d'index IY pointe sur la zone source. HL et DE sont des pointeurs de destination.

Suppression

Pour supprimer un élément, il suffit de décaler les éléments qui le suivent dans la liste, aux adresses supérieures, de la longueur d'un élément. La longueur de la liste est décrémentée. C'est ce qu'illustre la figure 9.14.

Le programme correspondant (figure 9.16) en découle tout simplement. Il se nomme « DELETE », et occupe les emplacements mémoire de 015F à 0187. La figure 9.15 montre l'ordinogramme correspondant.

L'emplacement mémoire TEMPTR sert de pointeur temporaire sur l'élément à déplacer vers le haut.

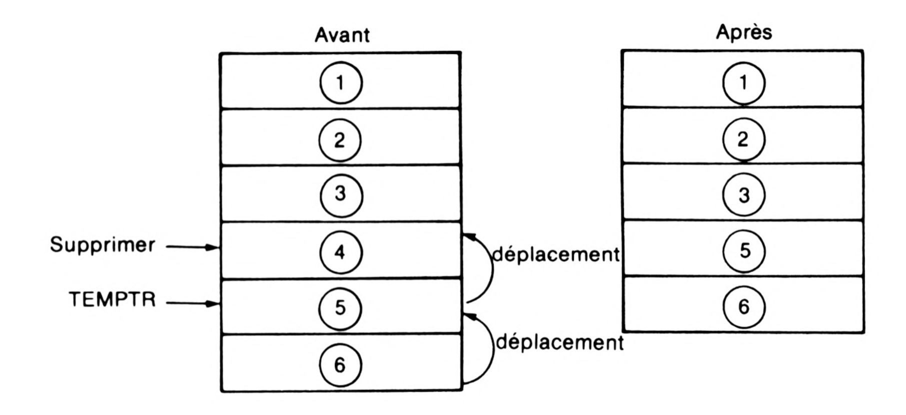


Figure 9.14. — Suppression d'une entrée (liste simple)

Durant le transfert, POINTR pointe toujours sur le « trou » dans la liste, c'est-à-dire sur la destination du prochain bloc.

L'indicateur Z signale, à la fin, la réussite de la suppression.

A noter l'utilisation de l'instruction LDIR pour le transfert, automatique et efficace, des blocs (cf. la ligne 0178 figure 9.16).

```
         LD    A, B           COMPTEUR DE BLOC
NEWBLOC  LD    BC, (ENTLEN)   LONGUEUR D'UN BLOC
         LDIR
         DEC   A
         JP    NZ, NEWBLOC
```

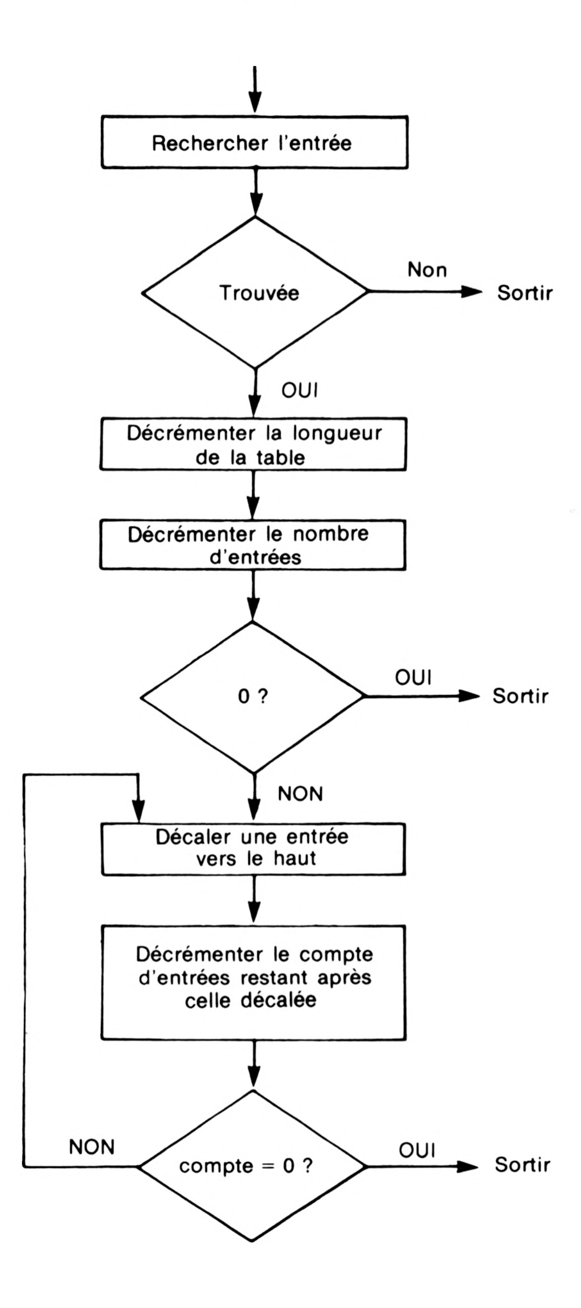


Figure 9.15. — Ordinogramme de suppression dans une table

```
0000                        ORG     0100H
        (0187)      ENTLEN  DL      ENDER
        (0189)      TABLEN  DL      ENDER+2
        (018A)      TABASE  DL      ENDER+3
        (018C)      TEMP    DL      ENDER+5
                    ;
0100    1600        SEARCH  LD      D,0             ;mettre D à zéro
0102    3A8901              LD      A,(TABLEN)      ;tester si la longueur de la table est nulle
0105    A7                  AND     A               ;positionner les indicateurs
0106    C8                  RET     Z
0107    47                  LD      B,A             ;ranger la longueur de la table
0108    DD2A8A01            LD      IX,(TABASE)     ;mettre l'adresse d'origine dans IX
010C    DD7E00      LOOP    LD      A,(IX+0)        ;tester la première lettre de l'entrée
010F    FDBE00              CP      (IY+0)
0112    C22701              JP      NZ,NEXTONE
0115    DD7E01              LD      A,(IX+1)        ; tester la seconde lettre
0118    FDBE01              CP      (IY+1)
011B    C22701              JP      NZ,NEXTONE
011E    DD7E02              LD      A,(IX+2)        ;tester la troisième lettre
0121    FDBE02              CP      (IY+2)
0124    CA3201              JP      Z,FOUND         ; sortir si toutes les lettres correspondent
0127    05          NEXTONE DEC     B               ; décrémenter le compteur de longueur
0128    C8                  RET     Z               ; sortir si fin de table
0129    ED5B8701            LD      DE,(ENTLEN)     ; positionner IX sur l'entrée suivante
012D    DD19                ADD     IX,DE
012F    C30C01              JP      LOOP            ; essayer encore
0132    16FF        FOUND   LD      D,0FFH          ; positionner D pour indiquer qu'IX
0134    C9                  RET                     ; ... contient l'adresse de l'entrée dans la table
                    ;
                    ;
                    ;
0135    CD0001      NEW     CALL    SEARCH          ; voir si déjà présent
0138    14                  INC     D
0139    CA5E01              JP      Z,OUTE          ; si D vaut FF, sortir
013C    3A8901              LD      A,(TABLEN)
013F    5F                  LD      E,A             ; charger E avec la longueur de la table
0140    3C                  INC     A
0141    328901              LD      (TABLEN),A      ; incrémenter la longueur de la table
0144    1600                LD      D,0
0146    2A8A01              LD      HL,(TABASE)
0149    ED4B8701            LD      BC,(ENTLEN)     ; charger B avec la longueur d'une entrée
014D    41                  LD      B,C
014E    19          LOOPE   ADD     HL,DE
014F    10FD                DJNZ    LOOPE           ; ajouter HL à (ENTLEN × TABLEN)
0151    ED4B8701            LD      BC,(ENTLEN)
0155    FDE5                PUSH    IY              ; mettre IY dans DE
0157    D1                  POP     DE
0158    EB                  EX      DE,HL
0159    EDB0                LDIR                    ; déplacer la mémoire de OBJET
015B    01FFFF              LD      BC,0FFFFH       ; ... en fin de table
015E    C9          OUTE    RET
                    ;
                    ;
                    ;
015F    CD0001      DELETE  CALL    SEARCH          ; chercher l'entrée à effacer
0162    14                  INC     D               ; tester si trouvée
0163    C28601              JP      NZ,OUT
0166    3A8901              LD      A,(TABLEN)      ; décrémenter la longueur de la table
0169    3D                  DEC     A
016A    328901              LD      (TABLEN),A
016D    05                  DEC     B               ; B = ≠ d'entrées restant dans la table
016E    CA8301              JP      Z,EXIT          ; ... après celle effacée
0171    DDE5                PUSH    IX              ; mettre IX dans DE
0173    D1                  POP     DE
0174    2A8701              LD      HL,(ENTLEN)     ; positionner HL une entrée au-dessus de DE
0177    19                  ADD     HL,DE
0178    78                  LD      A,B             ; positionner le compteur de bloc
0179    ED4B8701    NEWBLOC LD      BC,(ENTLEN)     ; et celui de longueur de bloc
017D    EDB0                LDIR                    ; décaler 1 entrée de la table
017F    3D                  DEC     A
0180    C27901              JP      NZ,NEWBLOC      ; décaler un autre bloc
0183    01FFFF      EXIT    LD      BC,0FFFFH       ; indiquer que le travail est fait
0186    C9          OUT     RET
                    ;
0187    (0000)      ENDER   END
```

Figure 9.16. — Liste simple — les programmes

```
SYMBOL TABLE

DELETE  015F    ENDER   0187    ENTLEN  0187    EXIT    0183    FOUND   0132
LOOP    010C    LOOPE   014E    NEW     0135    NEWBLO  0179    NEXTON  0127
OUT     0186    OUTE    015E    SEARCH  0100    TABASE  018A    TABLEN  0189
TEMP    018C
```

Figure 9.16. — Liste simple — les programmes (suite)

```
                 affichage mémoire                                  liste des objets
                                                                    avec leurs emplacements
                                                                    mémoire
-DM300
0300  53 4F 4E 31 31 31 31 31-31 31 31 31 31 00 00 00   SON1111111111...
0310  44 41 44 32 32 32 32 32-32 32 32 32 32 00 00 00   DAD2222222222...
0320  4D 4F 4D 33 33 33 33 33-33 33 33 33 33 00 00 00   MOM3333333333...
0330  55 4E 43 34 34 34 34 34-34 34 34 34 34 00 00 00   UNC4444444444...
0340  41 4E 54 35 35 35 35 35-35 35 35 35 35 00 00 00   ANT5555555555...
0350  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0360  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0370  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................

-SY
Y=0000 300     positionner IY à 0300 H (pointeur à OBJET)

-G193/196

P=0196 0196'   exécuter « INSERT »

                                                                    configuration de la table après
                                                                    l'exécution du programme
-DM400
0400  53 4F 4E 31 31 31 31 31-31 31 31 31 31 00 00 00   SON1111111111...
0410  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0420  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0430  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0440  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 0C 00 00 00   ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................

-SY
Y=0300 310     positionner IY à 0310 H (OBJET suivant)

-G193/196

P=0196 0196'   exécuter « INSERT »

                                                                    configuration de la table après
                                                                    la seconde insertion
-DM400
0400  53 4F 4E 31 31 31 31 31-31 31 31 31 31 44 41 44   SON1111111111DAD
0410  32 32 32 32 32 32 32 32-32 32 00 00 00 00 00 00   2222222222......
0420  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0430  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0440  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................

          . . .  autres insertions          . . .
                                                                    configuration de la table après
                                                                    plusieurs insertions
-DM400
0400  53 4F 4E 31 31 31 31 31-31 31 31 31 31 44 41 44   SON1111111111DAD
0410  32 32 32 32 32 32 32 32-32 32 55 4E 43 34 34 34   2222222222UNC444
0420  34 34 34 34 34 34 34 4D-4F 4D 33 33 33 33 33 33   4444444MOM333333
0430  33 33 33 33 41 4E 54 35-35 35 35 35 35 35 35 35   3333ANT555555555
0440  35 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   5...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

Figure 9.17. — Liste simple — un exemple d'exécution

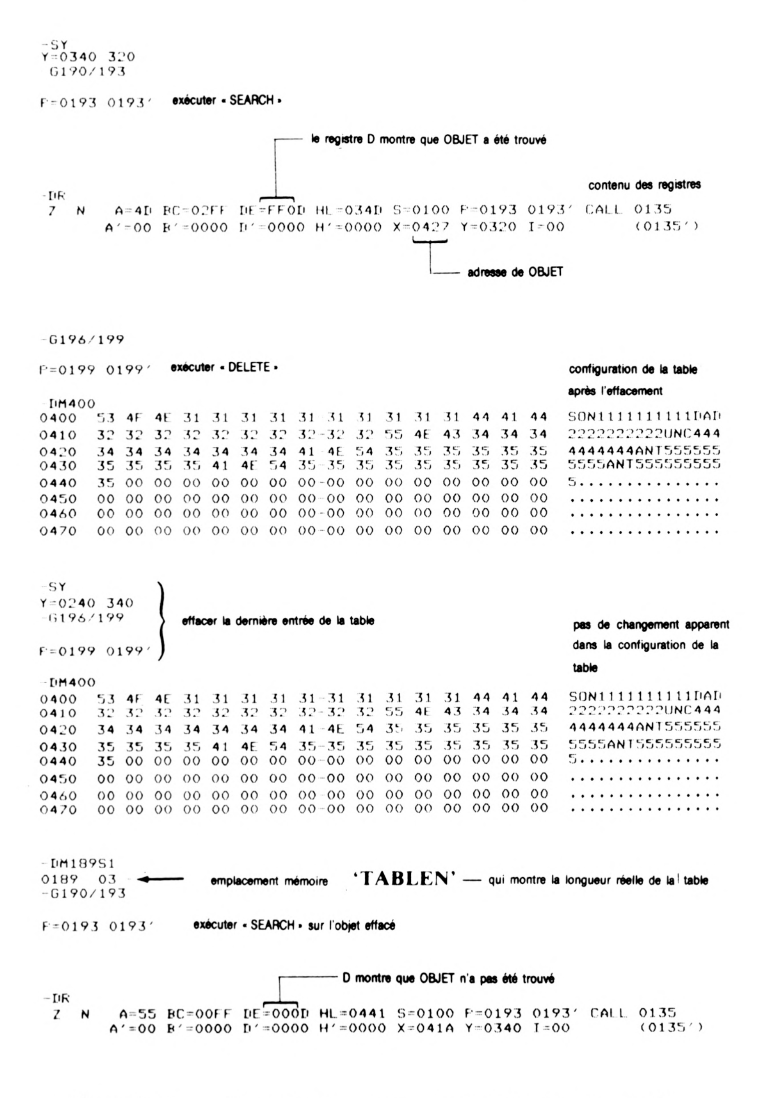


Figure 9.17. — Liste simple — un exemple d'exécution (suite)

LISTE ALPHABÉTIQUE

La liste alphabétique, ou « table », contrairement à la précédente, maintient tous ses éléments triés par ordre alphabétique, ce qui autorise l'utilisation de techniques de recherche plus rapides que la méthode linéaire. Exemple : la recherche dichotomique.

Recherche

Nous utiliserons l'algorithme dichotomique standard. Rappelons que cette technique est analogue à celle utilisée pour trouver un nom dans un annuaire téléphonique. L'opération consiste, habituellement, à choisir une page vers le milieu de l'annuaire et, selon les noms trouvés à cet endroit, à remonter vers la fin ou le début, jusqu'à la rencontre du nom recherché. Cette méthode est rapide, et assez simple à mettre en œuvre.

L'ordinogramme de recherche dichotomique apparaît à la figure 9.18, et le programme, à la figure 9.23.

Cette liste maintient les entrées en ordre alphabétique, et les retrouve au moyen d'une recherche dichotomique, ou « logarithmique » (cf. figure 9.19). La recherche est quelque peu compliquée par la nécessité de mémoriser plusieurs conditions. La principale erreur à éviter est la recherche d'un élément absent. Dans ce cas, les entrées de valeur alphabétique immédiatement inférieures et supérieures seraient testées, à tour de rôle, éternellement. Il faut donc maintenir, dans le programme, un indicateur qui préserve la valeur de l'indicateur de report, après une comparaison infructueuse. Lorsque la valeur de INCMNT, qui indique de combien le pointeur doit être incrémenté, atteint la valeur « 1 », un autre indicateur appelé « CLOSE NOW » (abrégé en « CLOSE »), est mis à la valeur de l'indicateur « COMPRES ». Ainsi, dans la mesure où tous les incréments suivants seront de « 1 », si le pointeur dépasse l'endroit où l'élément devrait se trouver, COMPRES ne sera plus égal à CLOSE, et la recherche se terminera. Ce mécanisme permet également au programme NEW de déterminer l'emplacement des pointeurs logique et physique, par rapport à la place théorique de l'élément.

Ainsi, si l'objet recherché n'est pas dans la table, et que le pointeur courant est incrémenté de 1, l'indicateur CLOSE sera positionné. A la prochaine itération du programme, le résultat de la comparaison sera l'opposé du précédent. Les deux indicateurs ne seront plus égaux, et le programme se terminera en indiquant : « non trouvé ».

Autre problème important : le franchissement de l'une des limites de la table, lors de l'addition ou de la soustraction de l'incrément. Il sera résolu en exécutant une « addition », ou une « soustraction », d'essai, à l'aide du pointeur logique et de la valeur contenant le nombre réel d'entrées, et non des positions, en mémoire, utilisées par les pointeurs physiques.

Le second indicateur utilisé par le programme est CLOSE. Il est mis à la valeur COMPRES quand l'incrément de recherche INCMNT devient égal à « 1 ». Il détectera le fait que l'élément n'a pas été trouvé, si COMPRES n'est pas égal à CLOSE au coup suivant.

Les autres variables utilisées sont :

LOGPOS qui indique la position logique dans la table (numéro de l'élément).

INCMNT, qui représente la valeur dont le pointeur courant sera incrémenté, ou décrémenté, si la prochaine comparaison échoue.

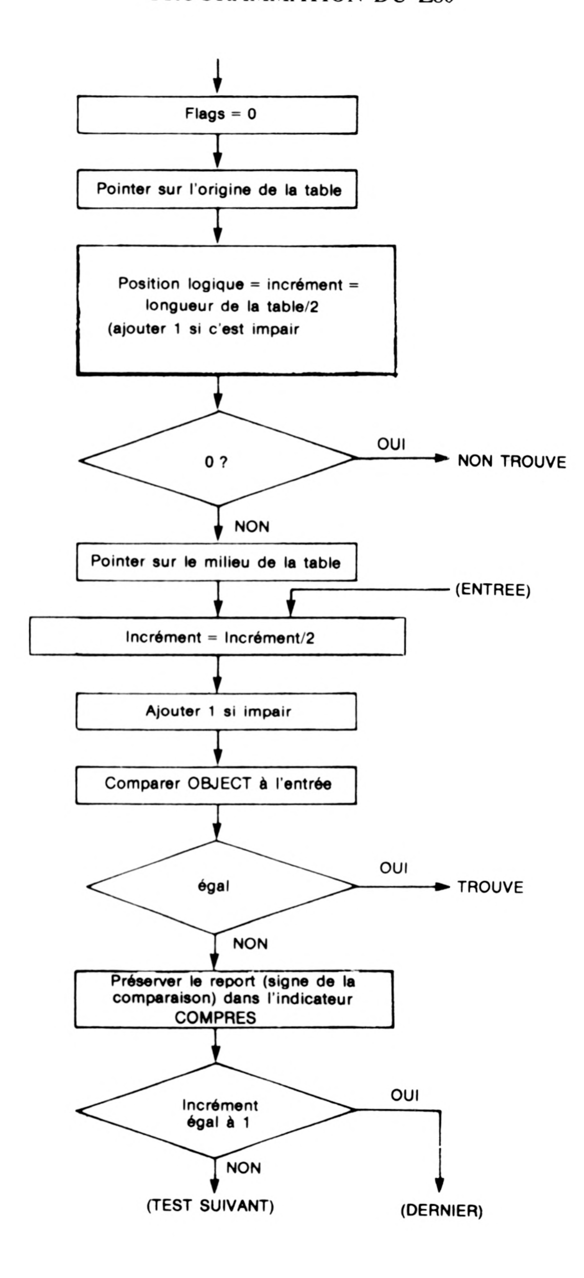


Figure 9.18. — Ordinogramme de recherche dichotomique

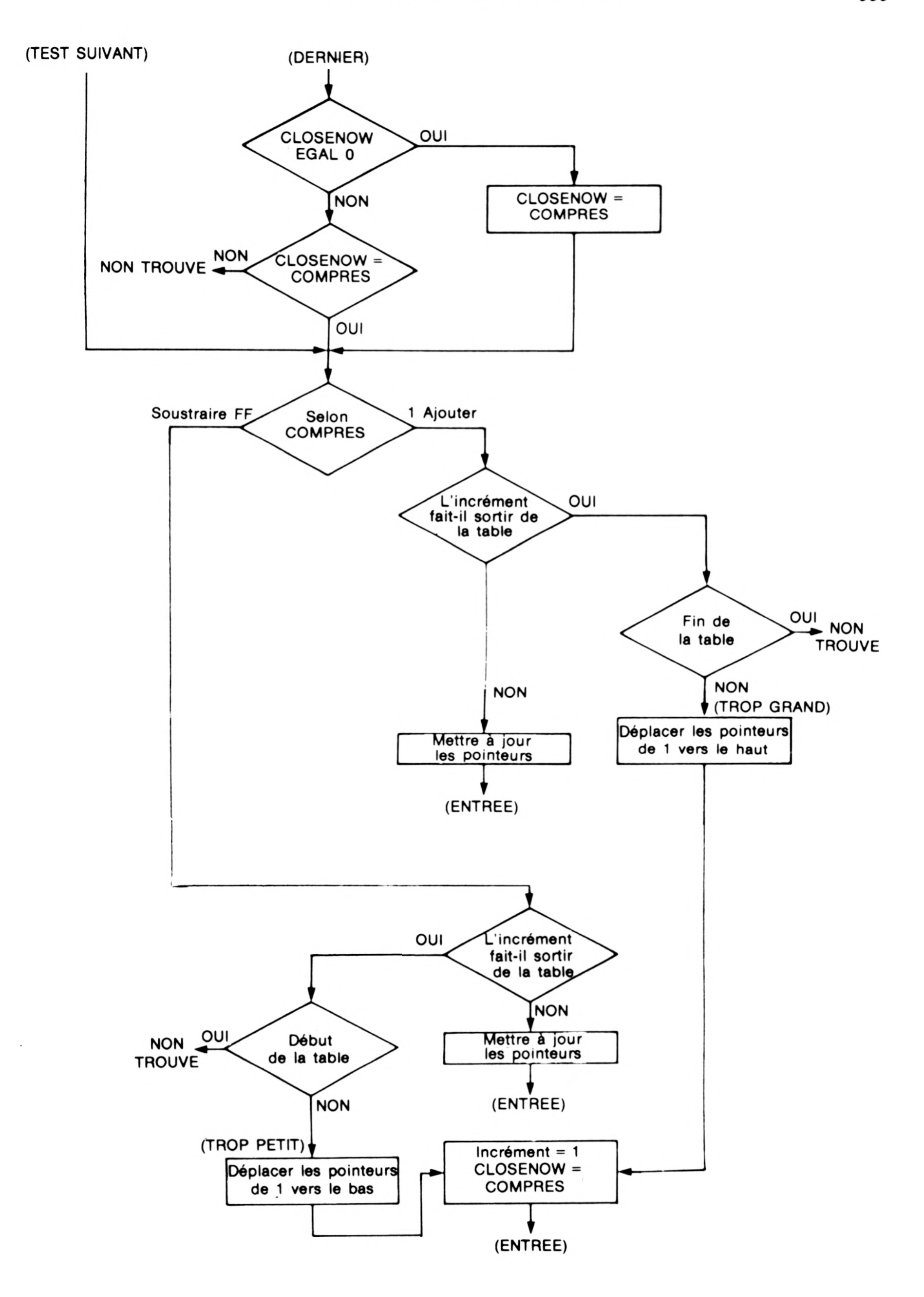


Figure 9.18. — Ordinogramme de recherche dichotomique (suite)

TABLEN représente, comme précédemment, la longueur totale de la liste. LOGPOS et INCMNT sont comparés à TABLEN, de façon à s'assurer que les limites de la liste ne sont pas franchies.

Le programme, nommé « SEARCH », est montré à la figure 9.23. Il occupe les emplacements mémoire de 0100 à 01CF, et mérite d'être étudié avec soin, car il est bien plus compliqué que dans le cas de la recherche linéaire.

Ce surcroît de complexité vient du fait que l'intervalle de recherche peut être tantôt pair ou impair. Dans le second cas, il est nécessaire d'introduire une correction. (Impossibilité, par exemple, de pointer sur l'élément du milieu d'une liste de quatre éléments.) Une « astuce » est employée pour pointer sur l'élément médian : la division par 2 est obtenue au moyen d'un décalage à droite. Le bit qui « tombe » dans le report, après l'instruction SRL, sera un « 1 » si l'intervalle est impair. Il sera alors, tout bêtement, ajouté au pointeur.

L'élément recherché, OBJECT, est comparé à l'entrée médiane du nouvel intervalle de recherche. Si la comparaison réussit, le programme est terminé. Sinon (« NOGOOD »), le report est mis à « 0 » si OBJECT est inférieur à l'entrée. Lorsque l'incrément INCMNT devient « 1 », l'indicateur CLOSE, (qui avait été initialisé à « 0 ») est alors testé pour savoir s'il est positionné. S'il ne l'est pas, il le devient. S'il l'est, une vérification est effectuée pour déterminer si nous avons passé l'endroit où OBJECT aurait dû se trouver, mais où il n'est pas.

```
(0121)      LD      A, C
            SRL     A
            ADC     O
            LD      C, A
```

En résumé, le programme utilise deux indicateurs pour conserver l'information : CMPRES et CLOSE. L'indicateur CMPRES permet de se souvenir si le report valait « 0 » ou « 1 », après la dernière comparaison, et donc de déterminer si l'élément testé était supérieur ou inférieur à l'objet de la comparaison. Le report C indique la relation. Chaque fois qu'il vaut « 1 », et que l'élément testé est inférieur à l'objet cherché, CMPRES est mis à « 1 ». Chaque fois qu'il vaut « 0 », et que l'élément est supérieur à l'objet, CMPRES est mis à « FF ».

Remarquons que lorsque le report vaut « 1 », le pointeur courant pointe l'entrée en dessous de l'OBJECT.

Insertion d'un élément

Pour insérer un nouvel élément, il convient, tout d'abord, d'effectuer une recherche dichotomique. Si l'élément est dans la table, il n'est pas nécessaire de l'insérer. (Nous supposons, ici, que tous les éléments sont distincts.) S'il y reste introuvable, il devra être inséré juste devant, ou juste derrière, le dernier élément auquel il a été comparé. Devant ? ou derrière ?

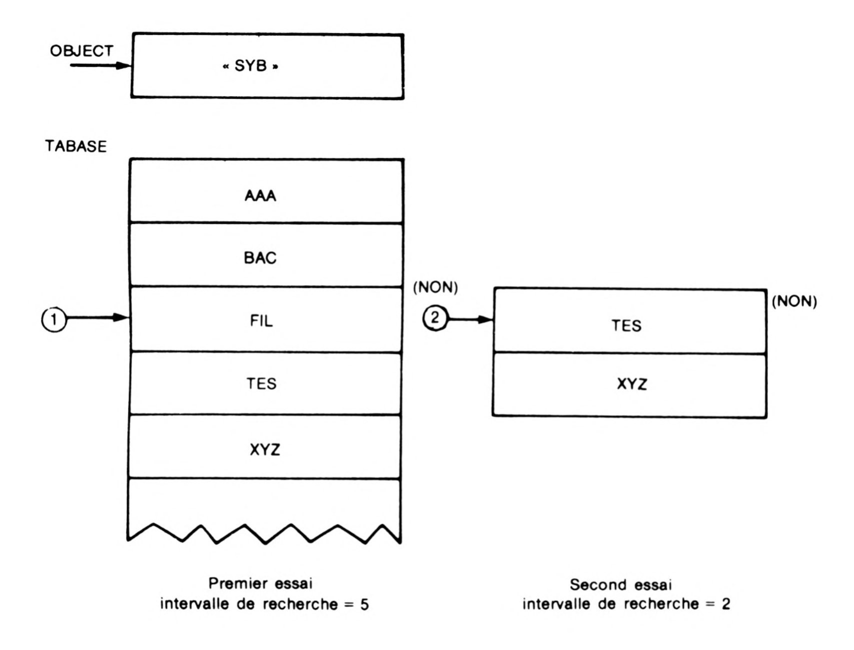


Figure 9.19. — Une recherche dichotomique

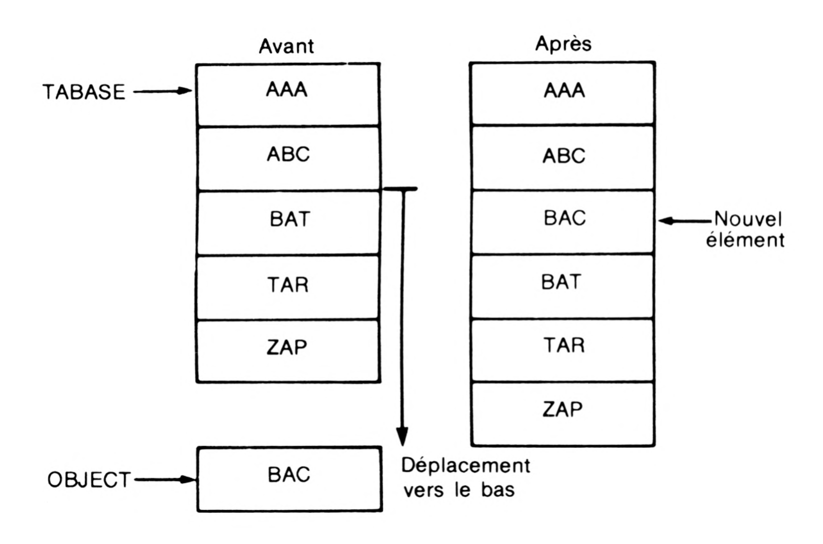


Figure 9.20. — Insertion de « BAC »

La valeur de l'indicateur COMPRES, après la recherche, le précise. Tous les éléments, à partir du futur emplacement de l'insertion, sont déplacés vers le bas.

Le processus d'insertion est illustré par la figure 9.20, et le programme correspondant apparaît figure 9.23.

Ce dernier s'appelle NEW, et commence à l'emplacement mémoire 0100. Notez que les instructions LDDR et LDIR du Z80 sont utilisées pour des transferts automatiques de bloc efficaces.

Suppression d'un élément

De la même façon, une recherche dichotomique est mise en œuvre pour découvrir l'objet. Si la recherche échoue, la suppression n'a aucune raison d'être effectuée. Si elle aboutit, l'élément est supprimé, et les suivants sont remontés d'une position de bloc (figure 9.21). Le programme apparaît figure 9.23, et l'ordinogramme figure 9.22.

Le programme s'appelle DELETE et commence à l'adresse 0221. La figure 9.24 donne un exemple d'exécution des programmes ci-dessus.

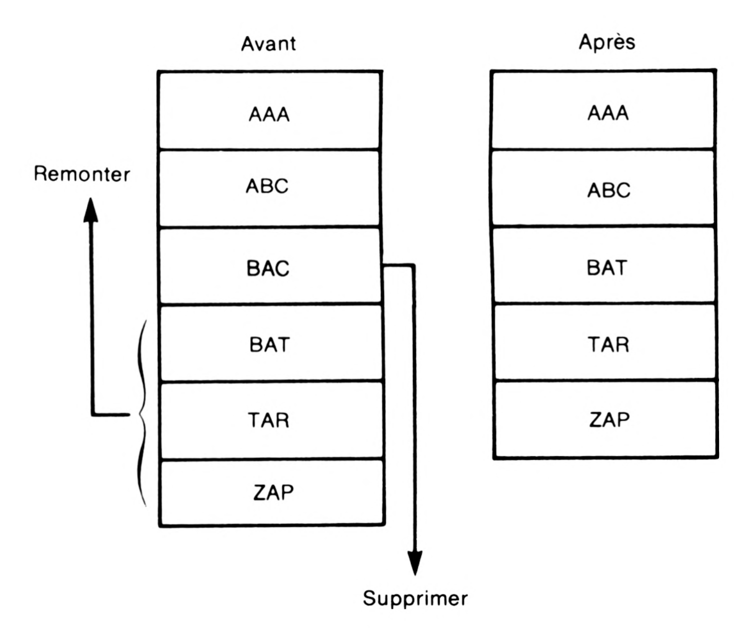


Figure 9.21. — Supprimer « BAC »

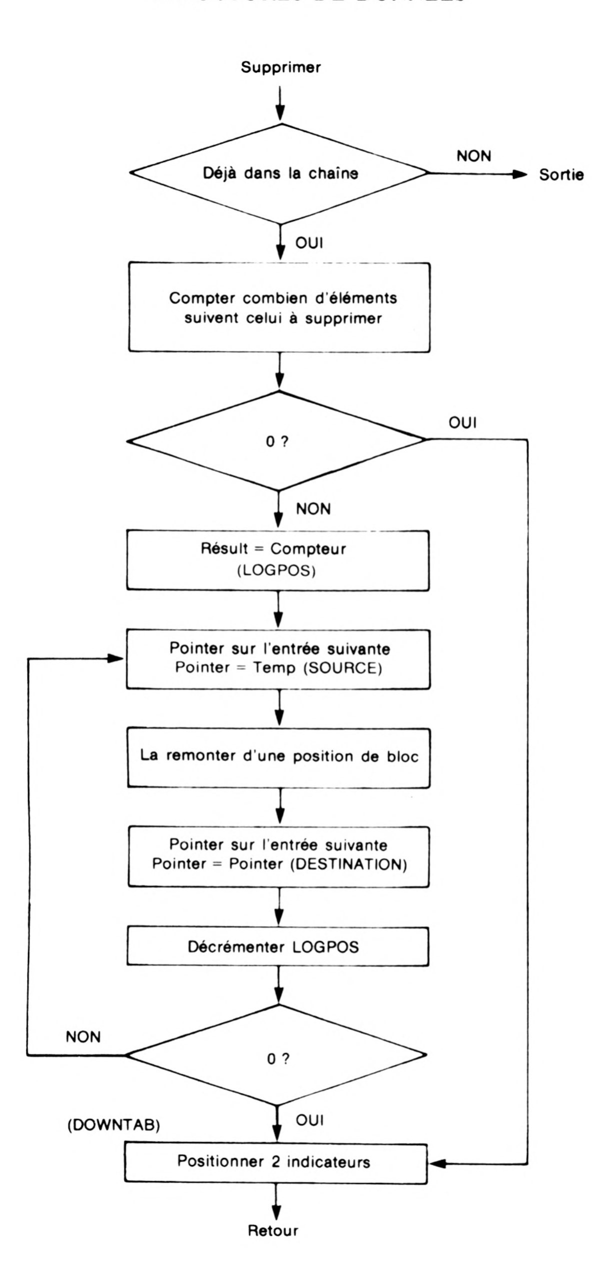


Figure 9.22. — Ordinogramme de suppression (liste alphabétique)

```
0000                              ORG    0100H
        (024A)    CLOSENOW  DL     ENDED
        (024B)    COMPRES   DL     ENDED+1
        (024C)    TABLEN    DL     ENDED+2
        (024D)    TABASE    DL     ENDED+3
        (024F)    ENTLEN    DL     ENDED+5
                  ;
0100    3E00      SEARCH    LD     A,0
0102    324A02              LD     (CLOSENOW),A     ; mettre à zéro les emplacements des indicateurs
0105    324B02              LD     (COMPRES),A
0108    57                  LD     D,A
0109    2A4D02              LD     HL,(TABASE)      ; initialiser HL
010C    3A4C02              LD     A,(TABLEN)
010F    CB3F                SRL    A                ; diviser par 2
0111    CE00                ADC    0                ; ajouter 1 si impair
0113    4F                  LD     C,A              ; ranger dans incrément
0114    47                  LD     B,A              ; et dans position logique
0115    CABA01              JP     Z,NOTFOUND       ; tester si longueur nulle
0118    5F                  LD     E,A              ; multiplier (E - 1) X ENTLEN
0119    1D                  DEC    E
011A    CDBD01              CALL   MULT
011D    19                  ADD    HL,DE            ; positionner HL sur le milieu de la table
011E    E5        ENTRY     PUSH   HL               ; charger HL dans IX
011F    DDE1                POP    IX
0121    79                  LD     A,C              ; diviser l'incrément par 2
0122    CB3F                SRL    A
0124    CE00                ADC    0
0126    4F                  LD     C,A
0127    DD7E00              LD     A,(IX+0)         ; comparer la première lettre
012A    FDBE00              CP     (IY+0)
012D    C24201              JP     NZ,NOGOOD
0130    DD7E01              LD     A,(IX+1)         ; comparer la deuxième lettre
0133    FDBE01              CP     (IY+1)
0136    C24201              JP     NZ,NOGOOD
0139    DD7E02              LD     A,(IX+2)         ; comparer la troisième lettre
013C    FDBE02              CP     (IY+2)
013F    CABC01              JP     Z,FOUND
0142    3E01      NOGOOD    LD     A,1              ; positionner l'indicateur du résultat de
0144    DA4901              JP     C,TESTS          ; ... la comparaison selon le résultat (1, FF)
0147    3EFF                LD     A,0FFH
0149    324B02    TESTS     LD     (COMPRES),A
014C    79                  LD     A,C              ; incrément = 1 ?
014D    3D                  DEC    A
014E    C26901              JP     NZ,NEXTEST
0151    3A4A02              LD     A,(CLOSENOW)     ; oui, indicateur CLOSE positionné ?
0154    A7                  AND    A
0155    CA6301              JP     Z,NOTCLOSE
0158    57                  LD     D,A              ; oui, voir si nous avons passé l'endroit
0159    3A4B02              LD     A,(COMPRES)      ; ... où l'entrée devrait être mais où elle n'est pas
015C    92                  SUB    D
015D    CA6901              JP     Z,NEXTEST
0160    C3BA01              JP     NOTFOUND
0163    3A4B02    NOTCLOSE  LD     A,(COMPRES)      ; positionner indicateur CLOSE selon le sens
0166    324A02              LD     (CLOSENOW),A     ; ... de la recherche pour éviter les répétitions
0169    DDE5      NEXTEST   PUSH   IX               ; préparer HL et DE pour une addition ou une
016B    E1                  POP    HL               ; ... soustraction selon l'incrément
016C    59                  LD     E,C
016D    CDBD01              CALL   MULT
0170    3A4B02              LD     A,(COMPRES)      ; addition ou soustraction ?
0173    3C                  INC    A
0174    C29601              JP     NZ,ADDIT
0177    78                  LD     A,B              ; tester si la soustraction fera franchir
0178    91                  SUB    C                ; la limite de la table
0179    CA8501              JP     Z,TOOLOW
017C    DA8501              JP     C,TOOLOW
017F    47                  LD     B,A              ; positionner la nouvelle position logique
0180    ED52                SBC    HL,DE            ; changer l'adresse elle-même
0182    C31E01              JP     ENTRY
0185    78        TOOLOW    LD     A,B              ; voir si la position est 1
0186    3D                  DEC    A
0187    CABA01              JP     Z,NOTFOUND       ; si c'est le cas sortir
018A    ED5B4F02            LD     DE,(ENTLEN)      ; soustraire juste la longueur d'une entrée
018E    37                  SCF
018F    3F                  CCF
0190    ED52                SBC    HL,DE
0192    05                  DEC    B                ; changer la position logique.
0193    C3AF01              JP     REALCLOS
```

Figure 9.23. — Programme de recherche dichotomique

```
0196  3A4C02    ADDIT    LD     A,(TABLEN)     ; vérifier que la position courante
0199  90                 SUB    B              ; ... plus l'incrément ne sera pas
019A  91                 SUB    C              ; ... hors de la table
019B  DAA501             JP     C,TOOHIGH
019E  19                 ADD    HL,DE          ; O.K changer l'adresse réelle
019F  78                 LD     A,B            ; changer la position logique
01A0  81                 ADD    C
01A1  47                 LD     B,A
01A2  C31E01             JP     ENTRY
01A5  81        TOOHIGH  ADD    C              ; voir si la position est au sommet de la
01A6  CABA01             JP     Z,NOTFOUND     ; ... table (comme pour TABLEN-B)
01A9  ED5B4F02           LD     DE,(ENTLEN)    ; ajouter 1 position
01AD  19                 ADD    HL,DE
01AE  04                 INC    B              ; incrémenter la position logique
01AF  0E01      REALCLOS LD     C,1            ; mettre l'incrément à 1
01B1  3A4B02             LD     A,(COMPRES)    ; positionner l'indicateur CLOSE selon
01B4  324A02             LD     (CLOSENOW),A   ; ... le résultat de la comparaison
01B7  C31E01             JP     ENTRY
01BA  16FF      NOTFOUND LD     D,0FFH
01BC  C9        FOUND    RET
                ;
01BD  E5        MULT     PUSH   HL             ; multiplier E par (ENTLEN)
01BE  C5                 PUSH   BC             ; résultat dans DE
01BF  1600               LD     D,0
01C1  210000             LD     HL,0000
01C4  ED4B4F02           LD     BC,(ENTLEN)
01C8  41                 LD     B,C
01C9  19        ADDEM    ADD    HL,DE
01CA  10FD               DJNZ   ADDEM
01CC  C1                 POP    BC
01CD  EB                 EX     DE,HL
01CE  E1                 POP    HL
01CF  C9                 RET
                ;
                ;
                ;
01D0  CD0001    NEW      CALL   SEARCH         ; voir si OBJET déjà présent
01D3  14                 INC    D
01D4  C22002             JP     NZ,OUT
01D7  3A4C02             LD     A,(TABLEN)     ; tester si table vide
01DA  A7                 AND    A
01DB  CA0C02             JP     Z,INSERT
01DE  3A4B02             LD     A,(COMPRES)
01E1  3C                 INC    A
01E2  CAED01             JP     Z,HISIDE
01E5  ED5B4F02           LD     DE,(ENTLEN)    ; COMPRES = 1, positionner HL au-dessus
01E9  19                 ADD    HL,DE          ; ... là où OBJET doit aller
01EA  C3EE01             JP     SETUP
01ED  05        HISIDE   DEC    B              ; COMPRES = 0, positionner B pour une soustraction
01EE  3A4C02    SETUP    LD     A,(TABLEN)     ; voir combien d'entrées restantes
01F1  90                 SUB    B
01F2  CA0C02             JP     Z,INSERT
01F5  5F                 LD     E,A            ; positionner HL sur la dernière position de la
01F6  CDBD01             CALL   MULT           ; ... dernière entrée
01F9  19                 ADD    HL,DE
01FA  2B                 DEC    HL
01FB  EB                 EX     DE,HL          ; positionner DE 1 entrée au-dessus de HL
01FC  2A4F02             LD     HL,(ENTLEN)
01FF  19                 ADD    HL,DE
0200  EB                 EX     DE,HL
0201  ED4B4F02  MOVEM    LD     BC,(ENTLEN)    , décaler une entrée vers le haut
0205  EDB8               LDDR
0207  3D                 DEC    A
0208  C20102             JP     NZ,MOVEM       ; répéter si nécessaire
020B  23                 INC    HL             ; HL pointe sur un emplacement maintenant vide
020C  FDE5      INSERT   PUSH   IY             ; charger OBJET dans l'espace libre
020E  D1                 POP    DE
020F  EB                 EX     DE,HL
0210  ED4B4F02           LD     BC,(ENTLEN)
0214  EDB0               LDIR
0216  3A4C02             LD     A,(TABLEN)     ; incrémenter la longueur de la table
0219  3C                 INC    A
021A  324C02             LD     (TABLEN),A
021D  01FFFF             LD     BC,0FFFFH      ; indiquer travail fait
0220  C9        OUT      RET
                ;
                ;
                ;
```

Figure 9.23. — Programme de recherche dichotomique (suite)

```
0221   CD0001     DELETE    CALL   SEARCH          ; chercher l'adresse de l'OBJET
0224   14                   INC    D               ; vérifier qu'il est présent
0225   CA4902               JP     Z,OUTE
0228   ED5B4F02             LD     DE,(ENTLEN)
022C   EB                   EX     DE,HL
022D   19                   ADD    HL,DE           ; DE pointe sur OBJET, HL une
022E   3A4C02               LD     A,(TABLEN)      ; ... entrée au-dessus
0231   90                   SUB    B               ; voir combien il reste d'entrées
0232   CA3F02               JP     Z,DOWNTAB
0235   ED4B4F02   SHIFTIN   LD     BC,(ENTLEN)
0239   EDB0                 LDIR                   ; décaler une entrée vers le haut
023B   3D                   DEC    A
023C   C23502               JP     NZ,SHIFTIN
023F   3A4C02     DOWNTAB   LD     A,(TABLEN)      ;décrémenter la longueur de la table
0242   3D                   DEC    A
0243   324C02               LD     (TABLEN),A
0246   01FFFF               LD     BC,0FFFFH       ; indiquer travail fait
0249   C9         OUTE      RET
                  ;
024A   (0000)     ENDED     END

SYMBOL TABLE

ADDEM    01C9     ADDIT    0196     CLOSEN   024A     COMPRE   024B     DELETE   0221
DOWNTA   023F     ENDED    024A     ENTLEN   024F     ENTRY    011E     FOUND    01BC
HISIDE   01ED     INSERT   020C     MOVEM    0201     MULT     01BD     NEW      01D0
NEXTES   0169     NOGOOD   0142     NOTCLO   0163     NOTFOU   01BA     OUT      0220
OUTE     0249     REALCL   01AF     SEARCH   0100     SETUP    01EE     SHIFTI   0235
TABASE   024D     TABLEN   024C     TESTS    0149     TOOHIG   01A5     TOOLOW   0185
```

Figure 9.23. — Programme de recherche dichotomique (suite)

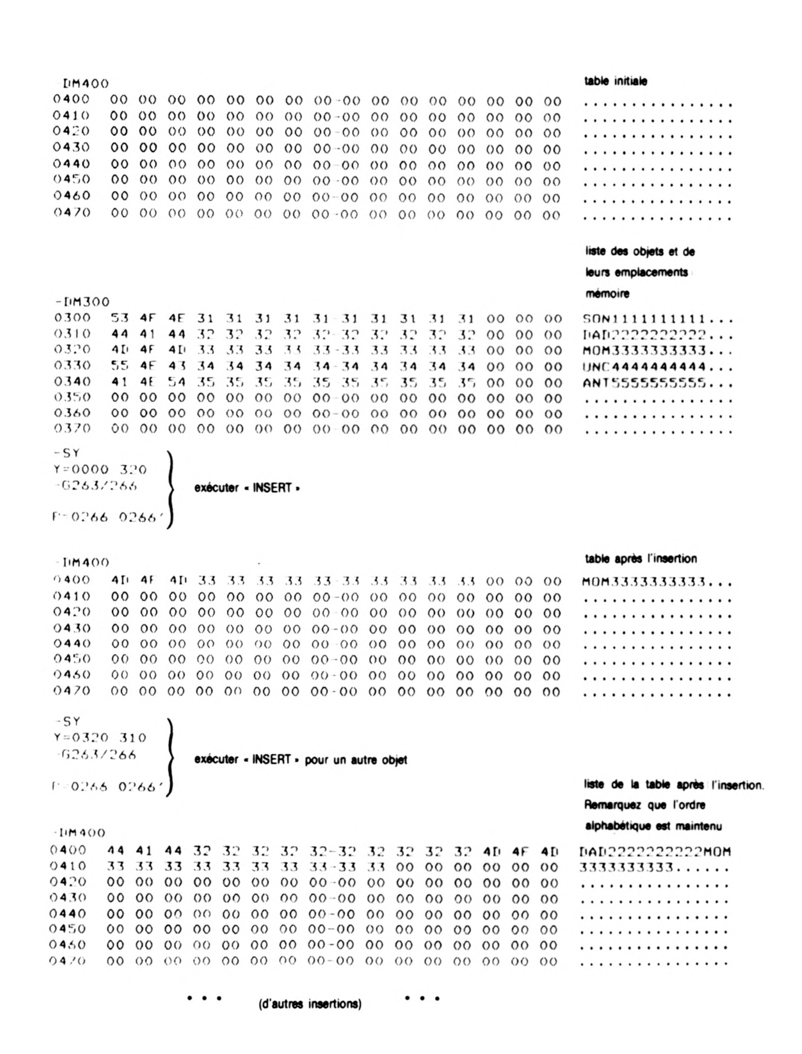
LISTE CHAÎNÉE

Nous supposerons que la liste chaînée contient, comme d'habitude, les trois caractères alphanumériques pour l'étiquette, suivis des octets de données (de 1 à 250), d'un pointeur sur deux octets contenant l'adresse de départ de l'entrée suivante, et enfin d'un octet de marquage. Chaque fois que cette marque est à « 1 », elle empêche la routine d'insertion de substituer une nouvelle entrée à celle qui existe.

En outre, un répertoire contient un pointeur sur la première entrée associée à chaque lettre de l'alphabet, de façon à faciliter la recherche. Nous supposerons, dans le programme, que les étiquettes sont des caractères alphabétiques ASCII. A la fin de la liste, tous les pointeurs sont mis à la valeur NIL, choisie ici égale à l'origine de la table, dans la mesure où cette valeur ne peut jamais advenir dans la liste chaînée.

Le programme d'insertion et de suppression effectue les manipulations évidentes du pointeur. Ils utilisent l'indicateur INDEXE pour signaler si un pointeur opérant sur un objet a pour source une entrée précédente de la liste, ou, au contraire, le répertoire. Les programmes correspondants sont à la figure 9.29, et la structure de données à la figure 9.25.

Application possible de cette structure de donnée : un annuaire géré par ordinateur, dans lequel chaque personne serait représentée par un code unique de trois lettres (peut-être ses initiales habituelles), le champ de données contenant, pour sa part, une adresse simplifiée, plus le numéro de téléphone (jusqu'à 250 caractères). Examinons cette structure plus en détail à la figure 9.23.



```
DM400                                                                 table initiale
0400  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0410  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0420  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0430  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0440  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................

                                                        liste des objets et de
                                                        leurs emplacements
-DM300                                                  mémoire
0300  53 4F 4E 31 31 31 31 31-31 31 31 31 31 00 00 00   SON1111111111...
0310  44 41 44 32 32 32 32 32-32 32 32 32 32 00 00 00   DAD2222222222...
0320  4D 4F 4D 33 33 33 33 33-33 33 33 33 33 00 00 00   MOM3333333333...
0330  55 4E 43 34 34 34 34 34-34 34 34 34 34 00 00 00   UNC4444444444...
0340  41 4E 54 35 35 35 35 35-35 35 35 35 35 00 00 00   ANT5555555555...
0350  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0360  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0370  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................

-SY          )
Y=0000 320   )
-G263/266    )  exécuter « INSERT »
             )
F=0266 0266' )

-DM400                                                  table après l'insertion
0400  4D 4F 4D 33 33 33 33 33-33 33 33 33 33 00 00 00   MOM3333333333...
0410  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0420  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0430  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0440  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................

-SY          )
Y=0320 310   )
-G263/266    )  exécuter « INSERT » pour un autre objet
             )
F=0266 0266' )                                          liste de la table après l'insertion.
                                                        Remarquez que l'ordre
-DM400                                                  alphabétique est maintenu
0400  44 41 44 32 32 32 32 32-32 32 32 32 32 4D 4F 4D   DAD2222222222MOM
0410  33 33 33 33 33 33 33 33-33 33 00 00 00 00 00 00   3333333333......
0420  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0430  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0440  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................

        . . .     (d'autres insertions)     . . .
```

Figure 9.24. — Liste alphabétique — un exemple d'exécution

configuration de la table après que tous les objets aient été insérés

```
-DM400
0400  41 4E 54 35 35 35 35 35-35 35 35 35 35 44 41 44  ANT5555555555DAD
0410  32 32 32 32 32 32 32 32-32 32 4D 4F 4D 33 33 33  2222222222MOM333
0420  33 33 33 33 33 33 33 53-4F 4E 31 31 31 31 31 31  3333333SON111111
0430  31 31 31 31 55 4E 43 34-34 34 34 34 34 34 34 34  1111UNC444444444
0440  34 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  4...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
-SY
Y=0340 300
-G260/263
P=0263 0263'
```

exécuter « SEARCH » pour « SON » (à l'adresse 0300)

trouvé

```
-DR
Z  N   A=4E BC=0401 DE=000D HL=0427 S=0100 P=0263 0263' CALL 01D0
       A'=00 B'=0000 D'=0000 H'=0000 X=0427 Y=0300 I=00      (01D0')
```

adresse de l'OBJET dans la table

(vérifier dans la table ci-dessus que c'est « SON »)

```
-G266/269
P=0269 0269'
```

exécuter « DELETE » pour « SON »

configuration de la table après l'effacement. Remarquez que UNC a été décalé vers le haut. La dernière entrée UNC doit être ignorée.

```
-DM400
0400  41 4E 54 35 35 35 35 35-35 35 35 35 35 44 41 44  ANT5555555555DAD
0410  32 32 32 32 32 32 32 32-32 32 4D 4F 4D 33 33 33  2222222222MOM333
0420  33 33 33 33 33 33 33 55-4E 43 34 34 34 34 34 34  3333333UNC444444
0430  34 34 34 34 55 4E 43 34-34 34 34 34 34 34 34 34  4444UNC444444444
0440  34 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  4...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
-G260/263
P=0263 0263'
```

exécuter « SEARCH » à nouveau (pour « SON »)

pas trouvé

```
-DR
S  N   A=FE BC=0401 DE=FF0D HL=0427 S=0100 P=0263 0263' CALL 01D0
       A'=00 B'=0000 D'=0000 H'=0000 X=0427 Y=0300 I=00      (01D0')
-G263/266
P=0266 0266'
```

réinsérer l'objet « SON »

configuration actuelle de la table. Comparez à celle préalable à DELETE

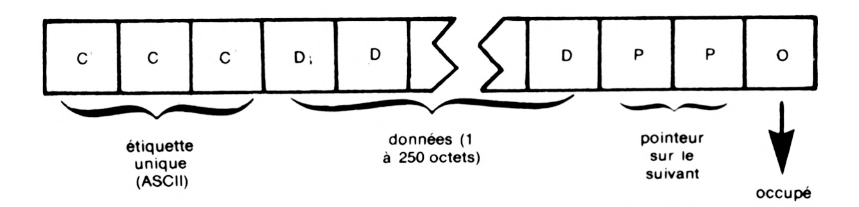
```
-DM400
0400  41 4E 54 35 35 35 35 35-35 35 35 35 35 44 41 44  ANT5555555555DAD
0410  32 32 32 32 32 32 32 32-32 32 4D 4F 4D 33 33 33  2222222222MOM333
0420  33 33 33 33 33 33 33 53-4F 4E 31 31 31 31 31 31  3333333SON111111
0430  31 31 31 31 55 4E 43 34-34 34 34 34 34 34 34 34  1111UNC444444444
0440  34 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  4...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

indique que le travail a été fait

```
-DR
       A=05 BC=FFFF DE=0434 HL=030D S=0100 P=0266 0266' CALL 0221
       A'=00 B'=0000 D'=0000 H'=0000 X=0427 Y=0300 I=00      (0221')
```

Figure 9.24. — Liste alphabétique. Un exemple d'exécution (suite)

Le format d'une entrée est :



Comme d'habitude, les conventions sont les suivantes :

ENTLEN longueur totale d'un élément (en octets)
TABASE adresse d'origine de la liste.

Nous supposerons toujours que l'adresse de OBJECT se trouve dans le registre IY, avant l'entrée dans le programme. Ici, REFBASE pointe sur l'adresse de début du répertoire, ou « table de référence ».

Dans le répertoire, chaque adresse sur deux octets pointe sur la première occurrence de la lettre à laquelle elle correspond dans la liste. Ainsi, chaque groupe d'entrées, dont les premières lettres du label sont identiques, forme réellement une liste indépendante dans la structure globale. Cette organisation, analogue à celle d'un carnet d'adresses, facilite la recherche. A noter qu'aucune donnée n'est déplacée lors d'une insertion ou d'une suppression. Seuls, les pointeurs sont changés, comme il arrive dans toute structure de liste chaînée fonctionnant correctement.

Si aucune entrée commençant par une lettre donnée n'est trouvée, ou si aucune entrée n'en suit une autre dans l'ordre alphabétique, leurs pointeurs feront référence au début de la table (= « NIL »). A la fin de la table, nous placerons, par convention, une valeur telle que la valeur absolue de la différence entre elle et « Z », soit plus grande que la différence entre « A » et « Z ». Elle représente la marque de fin de table (End of Table = EOT).

La valeur EOT sera supposée occuper la même taille mémoire qu'une entrée normale. Mais elle pourrait, le cas échéant, occuper juste un octet. Les lettres seront supposées être alphabétiques, et codées en ASCII. Modifier cet état de chose nécessiterait de changer la constante dans la routine PRETAB.

Le début de la table (« NIL ») est pris comme valeur de la marque de fin de table.

Par convention, les « pointeurs NIL », situés à la fin d'une chaîne ou dans une entrée du répertoire ne pointent sur aucune chaîne. Ils sont mis à la valeur de l'origine de la table, pour fournir un moyen d'identification unique. Une autre convention pourrait être choisie. Il résulte, en particulier, un gain de place de l'utilisation d'une marque EOT différente, dans la mesure où il n'est pas nécessaire de garder des entrées NIL pour les entrées inexistantes.

L'insertion et la suppression ont lieu de la manière habituelle (voir la première partie de ce chapitre), en modifiant simplement les pointeurs

nécessaires. L'indicateur INDEXE indique si le pointeur à l'objet concerné est dans le répertoire, ou dans un autre élément de la chaîne.

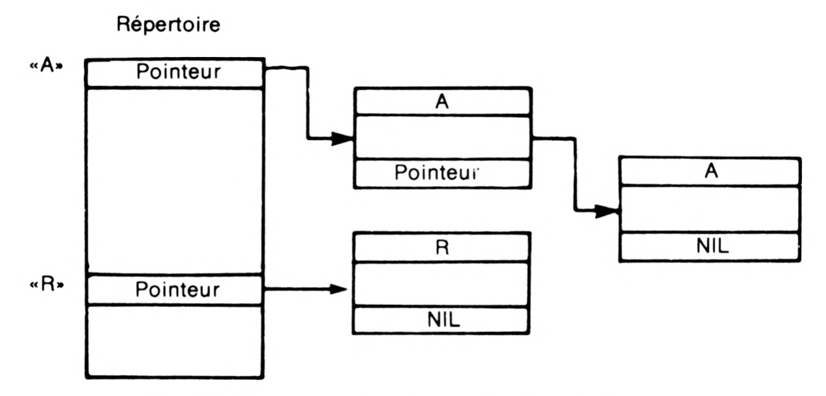


Figure 9.25. — Une structure de liste chaînée

Recherche

Le programme de recherche occupe les emplacements mémoire 0100 à 0155, et utilise le sous-programme PRETAB, situé à l'adresse 01D2.

Le principe de recherche est évident :

1. Accéder à l'entrée du répertoire correspondant à la lettre de l'alphabet figurant en première position de l'étiquette OBJECT.
2. Prendre le pointeur. Accéder à l'élément. S'il vaut NIL, l'entrée n'existe pas.
3. S'il ne vaut pas NIL, comparer l'élément à OBJECT, s'ils sont égaux, la recherche a abouti. S'ils ne le sont pas, prendre le pointeur sur l'entrée suivante de la liste.
4. Repartir en 2.

La figure 9.26 en donne un exemple.

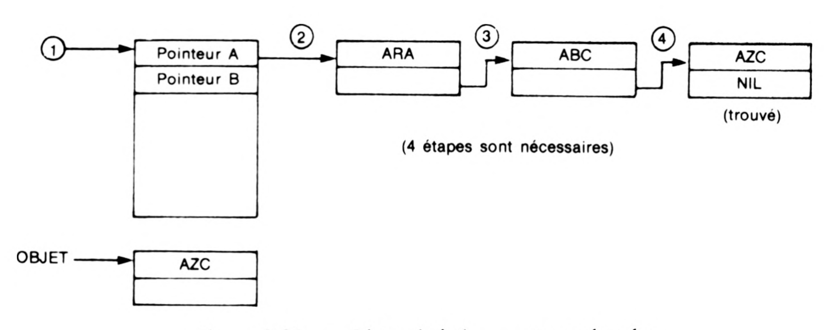


Figure 9.26. — Liste chaînée — une recherche

Insertion

L'insertion, c'est essentiellement une recherche, suivie d'une insertion, après la découverte de « NIL ».

Un bloc est alloué pour ranger la nouvelle entrée derrière la marque EOT, en cherchant une marque d'occupation à la valeur « disponible ».

Le programme s'appelle NEW [figure 9.29], et occupe les adresses 0156 à 0143.

La figure 9.27 donne un exemple.

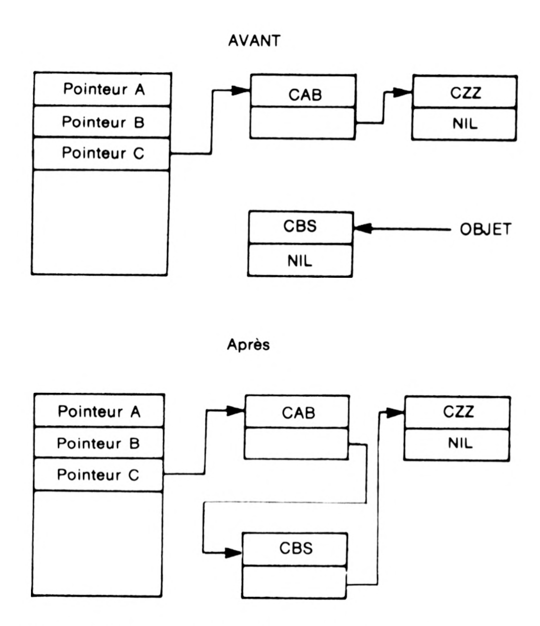


Figure 9.27. — Liste chaînée : exemple d'insertion

Suppression

L'élément sera supprimé en mettant sa marque d'occupation à la valeur « disponible », et en mettant à jour le pointeur dans le répertoire [ou dans l'élément pointant sur lui].

Le programme s'appelle « DELETE », et occupe les emplacements mémoire de 01A4 à 01D1.

La figure 9.28 donne un exemple de suppression.

RÉSUMÉ

Le programmeur débutant ne doit pas se sentir, dès maintenant, concerné par les détails de l'implantation et de la gestion des structures de données.

Toutefois, la programmation efficace d'algorithmes complexes nécessite une bonne compréhension de ces structures.

Les exemples concrets présentés dans ce chapitre aideront le lecteur à acquérir cette compréhension, et à résoudre tous les problèmes courants en faisant appel aux structures de données adaptées.

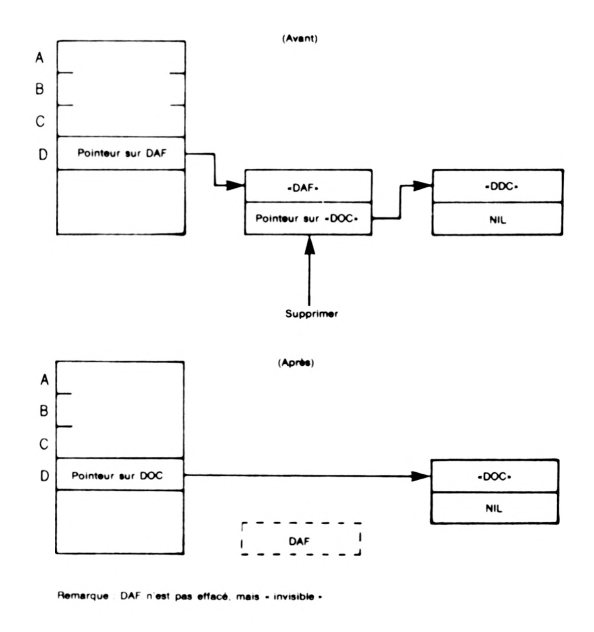
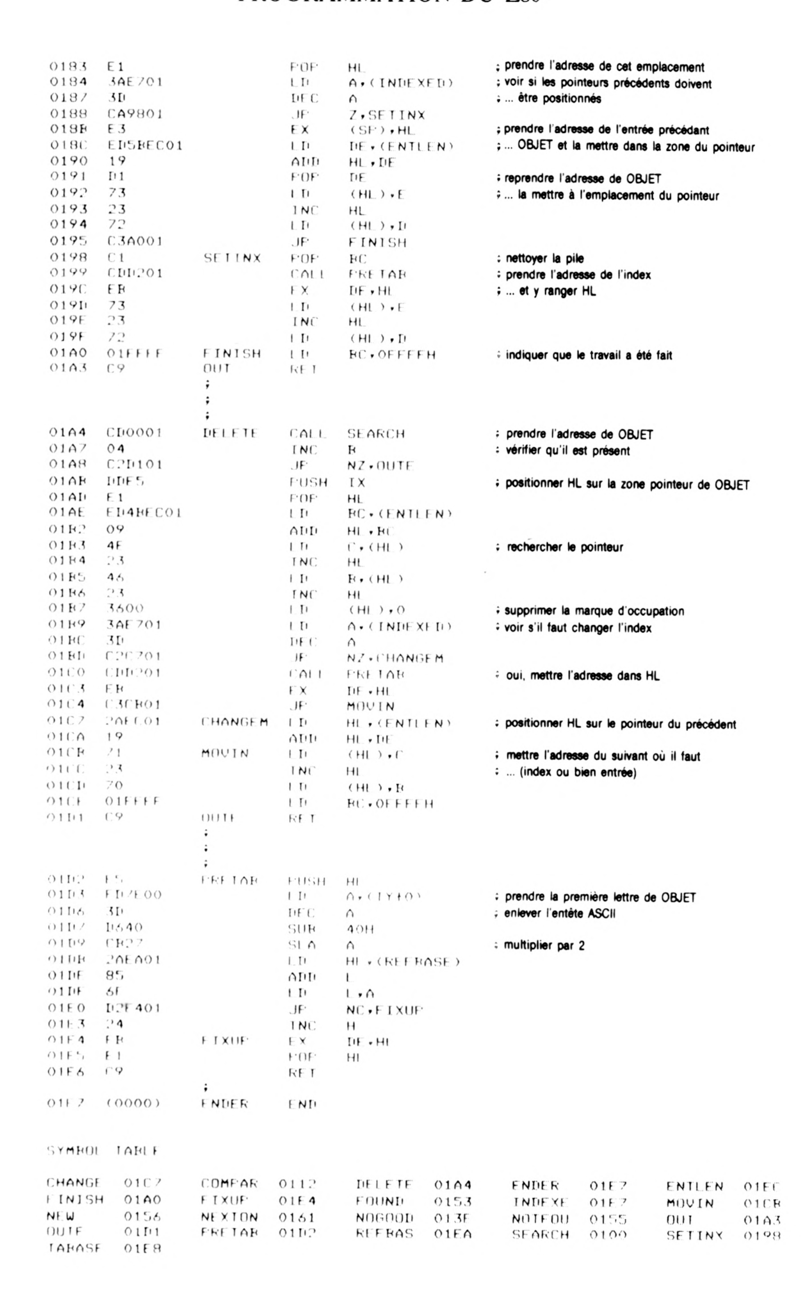


Figure 9.28. — Exemple de suppression (iiste chaînée)

```
0000'                              ORG     0100H
        (01E7)      INDEXED        DL      ENDER
        (01E8)      TABASE         DL      ENDER+1
        (01EA)      REFBASE        DL      ENDER+3
        (01EC)      ENTLEN         DL      ENDER+5
                    ;
0100    3E00        SEARCH         LD      A,0            ; initialiser les indicateurs
0102    47                         LD      B,A
0103    3C                         INC     A
0104    32E701                     LD      (INDEXED),A
0107    CDD201                     CALL    PRETAB         ; prendre l'adresse du pointeur d'index
010A    1A                         LD      A,(DE)         ; mettre le contenu du pointeur dans HL
010B    6F                         LD      L,A
010C    13                         INC     DE
010D    1A                         LD      A,(DE)
010E    67                         LD      H,A
010F    E5                         PUSH    HL
0110    DDE1                       POP     IX
0112    DD7E00      COMPARE        LD      A,(IX+0)       ; prendre la première lettre de l'entrée
0115    FE7C                       CP      7CH            ; voir si c'est une marque EOT
0117    D25501                     JP      NC,NOTFOUND
011A    DD7E00                     LD      A,(IX+0)       ; comparer les premières lettres
011D    FDBE00                     CP      (IY+0)
0120    DA3E01                     JP      C,NOGOOD
0123    C25501                     JP      NZ,NOTFOUND
0126    DD7E01                     LD      A,(IX+1)       ; comparer les secondes lettres
0129    FDBE01                     CP      (IY+1)
012C    DA3E01                     JP      C,NOGOOD
012F    C25501                     JP      NZ,NOTFOUND
0132    DD7E02                     LD      A,(IX+2)       ; comparer les troisièmes lettres
0135    FDBE02                     CP      (IY+2)
0138    CA5301                     JP      Z,FOUND
013B    D25501                     JP      NC,NOTFOUND
013E    DDE5        NOGOOD         PUSH    IX
0140    D1                         POP     DE
0141    2AEC01                     LD      HL,(ENTLEN)    ; aller au pointeur de l'entrée
0144    19                         ADD     HL,DE
0145    4E                         LD      C,(HL)         ; mettre la valeur du pointeur dans BC
0146    23                         INC     HL
0147    46                         LD      B,(HL)
0148    C5                         PUSH    BC             ; charger IX avec le pointeur
0149    DDE1                       POP     IX
014B    3E00                       LD      A,0
014D    32E701                     LD      (INDEXED),A    ; effacer l'indicateur
0150    C31201                     JP      COMPARE
0153    06FF        FOUND          LD      B,0FFH
0155    C9          NOTFOUND       RET
                    ;
                    ;
                    ;
0156    CD0001      NEW            CALL    SEARCH         ; voir où l'objet devrait aller
0159    04                         INC     B
015A    CAA301                     JP      Z,OUT
015D    D5                         PUSH    DE             ; ranger l'adresse de l'entrée précédente
015E    2AE801                     LD      HL,(TABASE)    ; chercher de la place dans la table
0161    EB          NEXTONE        EX      DE,HL          ; aller à la fin de l'entrée suivante
0162    2AEC01                     LD      HL,(ENTLEN)
0165    23                         INC     HL             ; ajouter 3 pour la longueur vraie de l'entrée
0166    23                         INC     HL
0167    23                         INC     HL
0168    19                         ADD     HL,DE
0169    7E                         LD      A,(HL)
016A    3D                         DEC     A
016B    CA6101                     JP      Z,NEXTONE      ; s'il y a quelque chose là, essayer à nouveau
016E    13                         INC     DE
016F    D5                         PUSH    DE             ; sauvegarder la position de la place vide
0170    FDE5                       PUSH    IY             ; mettre IY dans HL
0172    E1                         POP     HL
0173    ED4BEC01                   LD      BC,(ENTLEN)    ; mettre l'objet dans la table
0177    EDB0                       LDIR
0179    DDE5                       PUSH    IX             ; mettre l'adresse de l'entrée après objet
017B    E1                         POP     HL             ; ... dans l'emplacement du pointeur
017C    EB                         EX      DE,HL
017D    73                         LD      (HL),E
017E    23                         INC     HL
017F    72                         LD      (HL),D
0180    23                         INC     HL
0181    3601                       LD      (HL),1
```

Figure 9.29. — Liste chaînée — les programmes



```
0183  E1                     POP   HL             ; prendre l'adresse de cet emplacement
0184  3AE701                 LD    A,(INDEXED)    ; voir si les pointeurs précédents doivent
0187  3D                     DEC   A              ; ... être positionnés
0188  CA9801                 JP    Z,SETINX
018B  E3                     EX    (SP),HL        ; prendre l'adresse de l'entrée précédant
018C  ED5BEC01               LD    DE,(ENTLEN)    ; ... OBJET et la mettre dans la zone du pointeur
0190  19                     ADD   HL,DE
0191  D1                     POP   DE             ; reprendre l'adresse de OBJET
0192  73                     LD    (HL),E         ; ... la mettre à l'emplacement du pointeur
0193  23                     INC   HL
0194  72                     LD    (HL),D
0195  C3A001                 JP    FINISH
0198  C1        SETINX       POP   BC             ; nettoyer la pile
0199  CDD201                 CALL  PRETAB         ; prendre l'adresse de l'index
019C  EB                     EX    DE,HL          ; ... et y ranger HL
019D  73                     LD    (HL),E
019E  23                     INC   HL
019F  72                     LD    (HL),D
01A0  01FFFF    FINISH       LD    BC,0FFFFH      ; indiquer que le travail a été fait
01A3  C9        OUT          RET
                ;
                ;
                ;
01A4  CD0001    DELETE       CALL  SEARCH         ; prendre l'adresse de OBJET
01A7  04                     INC   B              ; vérifier qu'il est présent
01A8  C2D101                 JP    NZ,OUTE
01AB  DDE5                   PUSH  IX             ; positionner HL sur la zone pointeur de OBJET
01AD  E1                     POP   HL
01AE  ED4BEC01               LD    BC,(ENTLEN)
01B2  09                     ADD   HL,BC
01B3  4E                     LD    C,(HL)         ; rechercher le pointeur
01B4  23                     INC   HL
01B5  46                     LD    B,(HL)
01B6  23                     INC   HL
01B7  3600                   LD    (HL),0         ; supprimer la marque d'occupation
01B9  3AE701                 LD    A,(INDEXED)    ; voir s'il faut changer l'index
01BC  3D                     DEC   A
01BD  C2C701                 JP    NZ,CHANGEM
01C0  CDD201                 CALL  PRETAB         ; oui, mettre l'adresse dans HL
01C3  EB                     EX    DE,HL
01C4  C3CB01                 JP    MOVIN
01C7  2AEC01    CHANGEM      LD    HL,(ENTLEN)    ; positionner HL sur le pointeur du précédent
01CA  19                     ADD   HL,DE
01CB  71        MOVIN        LD    (HL),C         ; mettre l'adresse du suivant où il faut
01CC  23                     INC   HL             ; ... (index ou bien entrée)
01CD  70                     LD    (HL),B
01CF  01FFFF                 LD    BC,0FFFFH
01D1  C9        OUTE         RET
                ;
                ;
                ;
01D2  E5        PRETAB       PUSH  HL
01D3  FD7E00                 LD    A,(IY+0)       ; prendre la première lettre de OBJET
01D6  3D                     DEC   A              ; enlever l'entête ASCII
01D7  D640                   SUB   40H
01D9  CB27                   SLA   A              ; multiplier par 2
01DB  2AEA01                 LD    HL,(REFBASE)
01DE  85                     ADD   L
01DF  6F                     LD    L,A
01E0  D2E401                 JP    NC,FIXUP
01E3  24                     INC   H
01E4  EB        FIXUP        EX    DE,HL
01E5  E1                     POP   HL
01E6  C9                     RET
                ;
01E7  (0000)    ENDER        END


SYMBOL TABLE

CHANGE   01C7    COMPAR   0112    DELETE   01A4    ENDER    01E7    ENTLEN   01EC
FINISH   01A0    FIXUP    01E4    FOUND    0153    INDEXE   01E7    MOVIN    01CB
NEW      0156    NEXTON   0161    NOGOOD   013F    NOTFOU   0155    OUT      01A3
OUTE     01D1    PRETAB   01D2    REFBAS   01EA    SEARCH   0100    SETINX   0198
TABASE   01E8
```

Figure 9.29. — Liste chaînée — les programmes (suite)

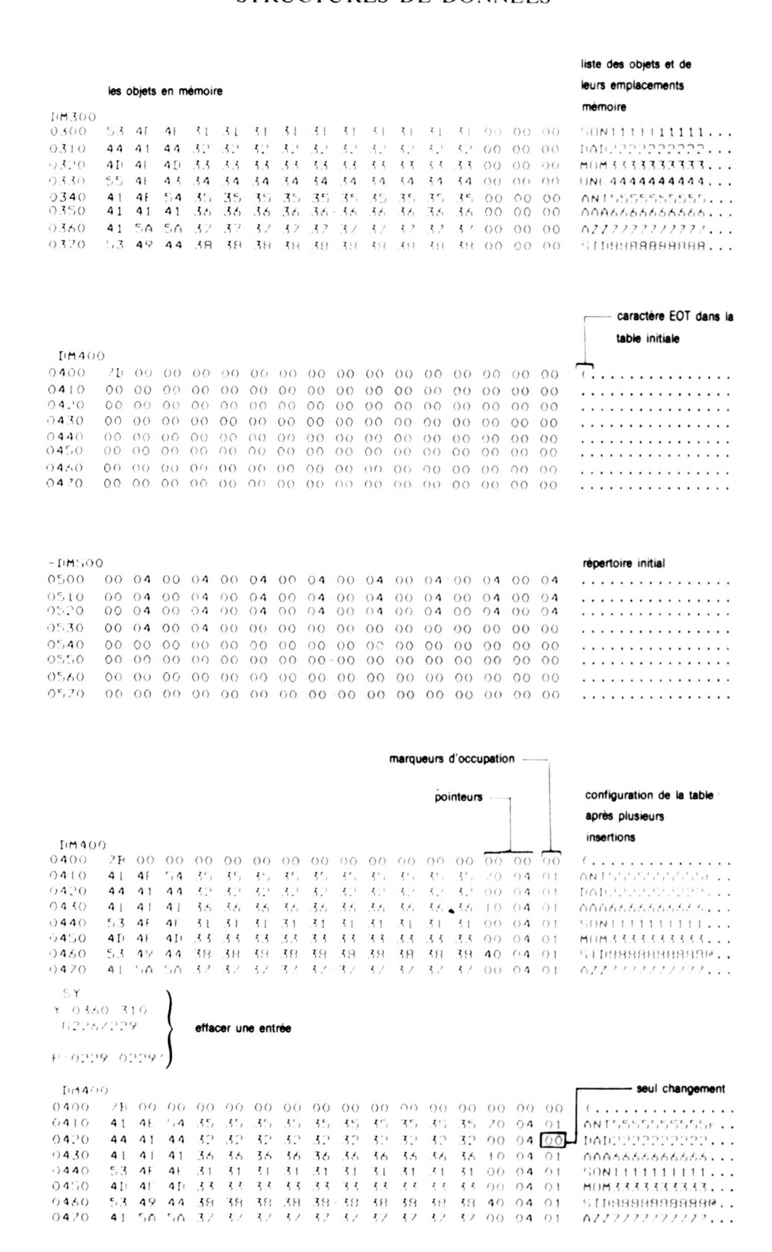


Figure 9.30. — Liste chaînée — un exemple d'exécution

```
-G220/223
                    exécuter « SEARCH » pour l'entrée supprimée
P=0223 0223'

                         — pas trouvée
-DR
    N   A=37 BC=00FF DE=0400 HL=0000 S=0100 P=0223 0223' CALL 0171
       A'=00 B'=0000 D'=0000 H'=0000 X=0400 Y=0310 I=00        (0171')

-SY
Y=0310 340
                    exécuter « SEARCH » pour une entrée existante
-G220/223

P=0223 0223'
                         — entrée trouvée
-DR
 Z  N   A=54 BC=FF10 DE=0430 HL=043E S=0100 P=0223 0223' CALL 0171
       A'=00 B'=0000 D'=0000 H'=0000 X=0410 Y=0340 I=00        (0171')
-G226/229                                          — adresse de l'entrée dans la table
                    effacer
P=0229 0229'


                                                              remarquez les changements
                                                              des pointeurs
-DM400
0400   7B 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   {...............
0410   41 4E 54 35 35 35 35 35-35 35 35 35 35 70 04 00   ANT5555555555p..
0420   44 41 44 32 32 32 32 32-32 32 32 32 32 00 04 00   DAD2222222222...
0430   41 41 41 36 36 36 36 36-36 36 36 36 36 70 04 01   AAA6666666666p..
0440   53 4F 4E 31 31 31 31 31-31 31 31 31 31 00 04 01   SON1111111111...
0450   4D 4F 4D 33 33 33 33 33-33 33 33 33 33 00 04 01   MOM3333333333...
0460   53 49 44 38 38 38 38 38-38 38 38 38 38 40 04 01   SID8888888888@..
0470   41 5A 5A 37 37 37 37 37-37 37 37 37 37 00 04 01   AZZ7777777777...
```

Figure 9.30. — Liste chaînée — un exemple d'exécution (suite)

10

LE DÉVELOPPEMENT DE PROGRAMMES

INTRODUCTION

Tous les programmes étudiés et développés jusqu'à présent, l'ont été à la main, sans l'aide d'une quelconque ressource logicielle ou matérielle. La seule amélioration apportée, par rapport au codage direct en binaire, est l'utilisation de symboles mnémoniques : ceux du langage assembleur. Pour développer efficacement du logiciel, il est nécessaire de connaître la gamme des outils logiciels et matériels d'aide au développement.

CHOIX FONDAMENTAUX DE PROGRAMMATION

Il existe trois modes essentiels d'écriture d'un programme : en binaire ou en hexadécimal, en langage de type assembleur, en langage de haut niveau. Examinons-les, tour à tour.

Codage hexadécimal

Le programme devrait être, normalement, écrit au moyen des mnémoniques du langage d'assemblage. Cependant, la plupart des systèmes informatiques monocartes de faible coût ne proposent pas d'assembleur. L'assembleur est un programme qui traduit automatiquement les mnémoniques dans les codes binaires requis. Lorsqu'aucun assembleur n'est disponible, cette traduction doit être faite à la main. Le binaire est *désagréable* à utiliser, et sujet à erreur. De sorte que l'hexadécimal lui est, généralement, préféré. Nous avons vu, au chapitre I, qu'un chiffre hexadécimal représente quatre chiffres binaires; et que deux chiffres hexadécimaux sont donc utilisés pour représenter le contenu de chaque octet. A titre d'exemple, la table donnant l'équivalent hexadécimal des instructions du Z80 figure en Appendice.

Pour résumer, chaque fois que les ressources de l'utilisateur sont limitées, et qu'aucun assembleur n'est disponible, le programme devra être codé à la main, en hexadécimal. Cette méthode peut être raisonnablement mise en œuvre pour un petit nombre d'instructions, de l'ordre de quelques dizaines. Pour les programmes plus importants, elle est fastidieuse et sujette à erreur. De sorte qu'il est préférable de l'éviter. Cependant, presque tous les microordinateurs en une seule carte nécessitent que les programmes soient entrés en hexadécimal. Ils ne sont pourvus ni d'assembleur, ni de clavier alphabétique complet, de façon à limiter leur coût.

En somme, le codage hexadécimal n'est pas la méthode appropriée à l'entrée d'un programme dans un ordinateur. Elle est simplement économique. Le coût d'un assembleur et d'un clavier alphanumérique contrebalance l'effort supplémentaire exigé. Cependant, cela ne change pas la façon dont le programme lui-même est écrit. *Le programme est malgré tout écrit en langage de type assembleur,* de façon à être accessible et compréhensible à un programmeur.

Programmation en langage assembleur

La programmation en langage de type assembleur s'applique à la fois aux programmes qui peuvent être entrés en hexadécimal, et à ceux qui peuvent l'être sous forme symbolique de type assembleur. Examinons ce dernier cas. Un programme d'assemblage doit être disponible. Il lit chaque instruction mnémonique du programme, et la traduit sous forme de la configuration de bits appropriée, en utilisant de 1 à 5 octets, comme l'exige le codage de l'instruction. Un bon assembleur offre, en outre, des facilités supplémentaires à l'écriture des programmes. Elles seront examinées plus loin [dans le paragraphe consacré à l'assembleur]. Des directives modifient notamment la valeur de symboles. L'adressage symbolique peut être employé pour désigner l'emplacement auquel il convient de se brancher. Lors de la phase de mise au point, au cours de laquelle le programmeur pourra enlever ou ajouter des instructions, il n'est pas nécessaire de réécrire tout le programme, dans le cas où une instruction supplémentaire est insérée entre un branchement et son lieu d'aboutissement. A condition toutefois que des étiquettes symboliques soient utilisées. L'assembleur prend soin d'ajuster automatiquement toutes les étiquettes, lors de la phase de traduction. De plus, il permet à l'utilisateur de mettre au point son programme sous forme symbolique. Un désassembleur permet, éventuellement, d'examiner le contenu d'un emplacement mémoire, et de reconstruire l'instruction dans le langage assembleur qu'il représente. Nous examinerons plus loin les diverses ressources logicielles normalement disponibles sur un système.

Examinons maintenant la troisième possibilité.

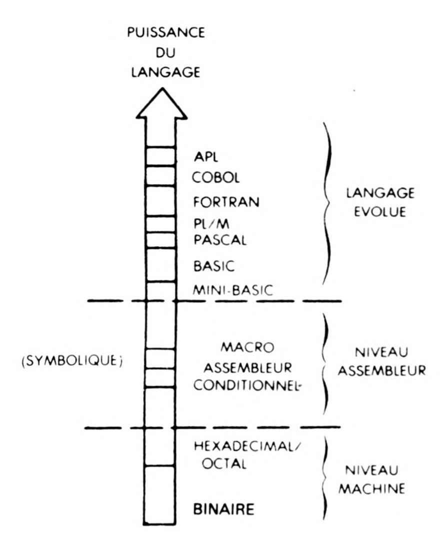


Figure 10.1. — Niveau de programmation

Langage de haut niveau

Un programme peut être écrit dans un langage de haut niveau, tel que BASIC, APL, PASCAL ou autres. Les techniques de programmation en ces divers langages sont traitées dans des ouvrages spécifiques. Nous n'avons pas ici l'intention de les passer en revue, et n'aborderons donc que brièvement ce mode de programmation. Un langage de haut-niveau procure des instructions puissantes, qui facilitent et accélèrent la programmation. Ces instructions doivent ensuite être traduites, par un programme complexe, dans le mode de représentation binaire que le microprocesseur pourra finalement exécuter. Chaque instruction de haut niveau sera, en règle générale, traduite en un grand nombre d'instructions binaires élémentaires. Le programme qui accomplit cette traduction automatique s'appelle un *compilateur*, ou un *interpréteur*. Un compilateur traduit successivement toutes les instructions d'un programme en code objet. Le code produit sera ensuite exécuté, dans une phase séparée. Au contraire, un interpréteur ne s'intéresse qu'à une seule instruction. Il l'interprète et l'exécute, puis « traduit » la suivante, l'exécute à son tour, et ainsi de suite. Un interprète offre l'avantage d'une réaction interactive, mais a pour conséquence une faible efficacité sur le plan de la vitesse, en comparaison d'un compilateur. Ces aspects ne seront pas étudiés plus longuement ici. Revenons maintenant à la programmation d'un microprocesseur concret en langage de type assembleur.

SUPPORT LOGICIEL

Nous allons passer en revue les principales facultés logicielles disponibles (ou qui devraient l'être) dans un système complet, pour la commodité du développement logiciel. Certaines définitions ont déjà été introduites, et nous nous contenterons ici de les résumer. Les autres programmes importants seront définis avant d'aller plus loin.

L'*assembleur* est le programme qui traduit les représentations mnémoniques des instructions en leur équivalent binaire. Il traduit une instruction symbolique en l'instruction binaire correspondante (qui peut occuper 1, 2 ou 3 octets). Le code binaire produit est appelé *code objet.* Il est directement exécutable par le microordinateur. En complément, l'assembleur produit un « listing » symbolique complet du programme, ainsi qu'une table d'équivalence, utilisable par le programmeur, et la liste d'occurence des symboles dans le programme. Nous en présenterons des exemples dans la suite de ce chapitre.

L'assembleur établit également la liste des erreurs de syntaxe : instructions interdites ou mal orthographiées, erreurs de branchement, étiquettes multiples ou omises.

Il ne supprime pas les erreurs de *logique* (c'est là *votre* problème).

Un *compilateur* est un programme qui traduit des instructions écrites en langage de haut niveau en leur équivalent binaire.

L'*interpréteur,* similaire au compilateur, traduit aussi des instructions de haut-niveau en leur équivalent binaire, mais il ne conserve pas la représentation de code intermédiaire. Il les exécute immédiatement. En fait, il ne génère souvent pas même de code intermédiaire, mais exécute plutôt directement les instructions de haut niveau.

Le *moniteur* est un programme élémentaire, indispensable pour utiliser les ressources matérielles du système. Il surveille continuellement ce qui arrive sur les organes d'entrée, et gère les autres éléments. Par exemple, un moniteur minimum pour microordinateur en une seule carte, équipé d'un clavier et de LED, doit scruter continuellement le clavier pour détecter une action éventuelle de l'utilisateur, et afficher les données spécifiées sur les diodes électroluminescentes. De plus, il doit être capable de comprendre un nombre restreint de commandes entrées au clavier, telles que DÉPART, ARRÊT, CONTINUER, CHARGEMENT MÉMOIRE, EXAMEN MÉMOIRE. Sur un système important, le moniteur est souvent appelé programme exécutif, lorsqu'il assure en plus l'enchaînement complexe de tâches ou la gestion sophistiquée de fichiers. L'ensemble de ces facultés est appelé *système d'exploitation.* Si les fichiers sont stockés sur disque, nous parlerons de *système d'exploitation disque,* ou DOS (Disk Operating System).

Un *« éditeur »* est un programme destiné à faciliter l'entrée et la modification de textes ou de programmes. Il permet à l'utilisateur d'entrer commodément des caractères, d'en ajouter, d'en insérer, d'ajouter des lignes, d'en enlever, de rechercher des caractères, ou des chaînes de caractères. C'est une ressource importante, très pratique et efficace pour entrer des textes.

L'épuration des programmes nécessite un programme de mise au point (« débugger »). Aucune indication n'est généralement fournie sur la cause du mauvais fonctionnement d'un programme. Le programmeur souhaite alors insérer des points d'arrêt, de façon à suspendre l'exécution du programme aux adresses désignées, et examiner alors le contenu des registres ou de la mémoire. C'est la fonction essentielle d'un « debugger ». Le « debugger » permet de suspendre un programme, d'en reprendre l'exécution, d'examiner, d'afficher et de modifier le contenu des registres et de la mémoire. Un bon « debugger » disposera d'un certain nombre de facultés supplémentaires. Il permettra notamment d'examiner les données sous forme symbolique, hexadécimale, binaire, ou autre, et d'entrer les données sous ces mêmes formats.

Un chargeur, ou chargeur éditeur de liens, place différents blocs de code objet aux emplacements mémoire indiqués, et ajuste leurs pointeurs symboliques respectifs de telle façon qu'ils puissent faire référence les uns aux autres. Il sert à reloger des programmes, ou des blocs, en diverses zones mémoire. Un simulateur, ou un émulateur, permet, pour sa part, de simuler lors du développement d'un programme le fonctionnement d'un élément, habituellement le microprocesseur, sans en disposer. Le programme est exécuté sur le processeur simulé avant d'être mis sur la carte réelle. Avec cette approche, il devient possible de suspendre le programme, de le modifier et de le garder en mémoire RAM. Les inconvénients d'un simulateur sont les suivants :

1. Il simule généralement le seul processeur, et non pas les organes d'entrée-sortie ;
2. La vitesse d'exécution est lente et le travail a lieu en temps simulé. Il n'est ainsi pas possible de tester les organes en temps réel. D'où de possibles problèmes de synchronisation, même si la logique du programme a été jugée correcte.

Un émulateur est essentiellement un simulateur en temps réel. Il utilise un processeur pour en simuler un autre. Et cela, dans tous ses détails.

Les routines utilitaires sont essentiellement les routines requises dans la plupart des applications, et dont l'utilisateur souhaite qu'elles soient fournies par le constructeur.

Nous y inclurons la multiplication, la division et autres opérations arithmétiques, les routines de déplacement de blocs, les tests de caractère, les routines gérant les organes d'entrée-sortie (« drivers »), bien d'autres encore.

LA SÉQUENCE DE DÉVELOPPEMENT D'UN PROGRAMME

Nous allons maintenant examiner une séquence typique de développement d'un programme en assembleur. De façon à démontrer leur intérêt, nous supposerons que toutes les facultés logicielles courantes sont disponibles. Si elles ne l'étaient pas, sur un système donné, il serait quand même

possible de développer des programmes, mais la commodité serait moindre, et le temps nécessaire à la mise au point vraisemblablement accru.

L'approche normale consiste, tout d'abord, à concevoir un algorithme, et à définir des structures de données pour le problème à résoudre. Ensuite, à développer un ensemble complet d'ordinogrammes représentant le déroulement du programme. Enfin, à traduire les algorithmes en langage assembleur du microprocesseur : c'est la phase de codage.

Le programme doit ensuite être entré dans l'ordinateur. Nous examinerons, au paragraphe suivant, les options matérielles à utiliser pour cette phase.

Le programme est entré dans la mémoire RAM du système sous le contrôle de l'éditeur. Une fois entrée une partie du programme [par exemple : un ou plusieurs sous-programmes], elle doit être testée.

Nous utiliserons d'abord l'assembleur. Si ce dernier ne se trouve pas déjà dans la mémoire du système, il sera chargé depuis un support extérieur, un disque, par exemple. Le programme est ensuite assemblé, autrement dit traduit en code binaire. Il en résulte un programme objet prêt à être exécuté.

Un programme n'est normalement pas censé marcher correctement dès la première fois. Pour vérifier son fonctionnement, on pose souvent un certain nombre de points d'arrêt, à des endroits cruciaux où il est facile de tester si les résultats intermédiaires sont corrects. On utilise, dans ce but, le « debugger ». Des points d'arrêt sont spécifiés à des endroits choisis, puis une commande « exécution » est émise pour démarrer l'exécution du programme. Ce dernier s'arrêtera, automatiquement, à chacun des points d'arrêt spécifiés. Le programmeur pourra alors vérifier, en examinant le contenu des registres ou de la mémoire, que les valeurs sont jusque là correctes. Dans ce cas, l'opération continuera jusqu'au point d'arrêt suivant. Chaque fois que des données incorrectes sont trouvées, une erreur du programme est détectée et doit donc être identifiée.

Pour celà, le programmeur se réfère normalement au listing de son programme, et vérifie la correction du codage. Si aucune erreur n'est décelée dans la programmation, il s'agira d'une erreur de logique, et il sera nécessaire de se référer à l'ordinogramme. Nous supposerons ici que les ordinogrammes ont été vérifiés à la main, et admettrons donc qu'ils sont suffisamment corrects. L'erreur a toute chance de venir du codage. Il est donc nécessaire de modifier une section du programme. Si la représentation symbolique est toujours en mémoire, nous entrerons à nouveau dans l'éditeur, modifierons les lignes concernées, puis réexécuterons la séquence précédente. Dans certains systèmes, la mémoire disponible est insuffisante, de sorte qu'il est nécessaire de vider la représentation symbolique du programme sur un disque, ou une cassette, avant d'exécuter le code objet. Dans ce cas, il faut naturellement recharger en mémoire la représentation symbolique du programme, à partir de son support, avant d'entrer à nouveau dans l'éditeur.

La procédure ci-dessus sera répétée aussi longtemps qu'il le faudra pour que les résultats du programme soient corrects. Rappelons l'adage : « mieux vaut prévenir que guérir ». Une conception correcte engendre généralement

une exécution elle-même correcte, dès lors que sont éliminées des habituelles fautes de frappe, et les erreurs de codage évidentes. A l'inverse, une conception baclée produit fréquemment des programmes très longs à mettre au point. Il est généralement admis que le temps de mise au point est plus long que celui de la conception proprement dite. En bref, il est toujours payant de consacrer plus de temps à la conception, de façon à raccourcir la phase de mise au point.

Toutefois, avec cette approche, il est possible de tester l'organisation générale du programme, mais pas en temps réel, avec les organes d'entrée-sortie. Si ces derniers doivent être testés, la solution directe réside dans le transfert du programme sur une EPROM, qui sera ensuite installée sur la carte. Il reste à vérifier que « ça marche ».

Il existe une meilleure solution : l'utilisation d'un *émulateur in-situ* (ICE : in-circuit emulator). Un ICE utilise le microprocesseur (Z80 ou autre) du système de développement pour émuler physiquement, en temps (presque) réel, un circuit Z80. L'émulateur est équipé d'un câble, terminé par un connecteur 40 broches, exactement identique au brochage du Z80. Le

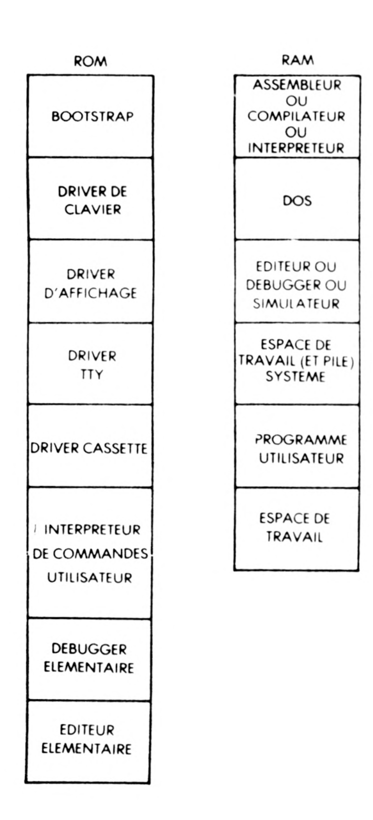


Figure 10.2. — Implantation mémoire typique

connecteur peut alors être inséré sur la vraie carte d'application. Les signaux générés par l'émulateur sont très exactement ceux du Z80, juste, peut-être, un peu plus lents. L'avantage essentiel est que le programme en cours de test réside toujours dans la mémoire RAM du système de développement. Il génère les signaux réels qui communiquent avec les vrais organes d'entrée-sortie utilisés. Ainsi, il devient possible de poursuivre le développement du programme, tout en mettant à contribution toutes les ressources du système de développement (éditeur, « debugger », utilisation du symbolique, gestion de fichiers), et en testant, en temps réel, les entrées-sorties.

Un bon émulateur dispose, en outre, de facultés spéciales, telles que la *trace*. La trace est un enregistrement des dernières instructions, ou du statut des divers bus de données dans un système, immédiatement avant un point d'arrêt. En bref, une trace fournit un film des événements survenus avant le point d'arrêt, ou la panne. Il est même possible de déclencher un oscilloscope sur l'accès à une adresse donnée, ou sur l'occurence d'une combinaison spécifique de bits. Cette possibilité est d'un grand intérêt. En effet, lorsqu'une erreur est trouvée, il est généralement trop tard. L'instruction ou la donnée, cause de l'erreur, est antérieure à la détection. L'existence d'une trace permet à l'utilisateur de retrouver le segment de programme fautif. Si la trace n'est pas assez longue, il suffit de poser un point d'arrêt en amont.

Ceci complète notre description de la séquence courante d'événements employée pour développer un programme. Passons maintenant en revue les possibilités matérielles de développement des programmes.

LES CHOIX MATÉRIELS

Microordinateur en une seule carte

Le microordinateur en une seule carte représente la façon la plus économique d'aborder le développement de programmes. Il est, normalement, équipé d'un clavier hexadécimal, de quelques touches de fonction, et de six afficheurs capables de visualiser adresse et donnée. Dans la mesure où un tel système ne possède qu'une petite quantité de mémoire, il ne dispose pas, le plus souvent, d'assembleur. Au mieux, il comporte un petit moniteur, et pratiquement, aucune facilité d'édition ou de mise au point, exception faite d'un tout petit nombre de commandes. Tous les programmes peuvent, cependant, être entrés sous forme hexadécimale. Ils peuvent aussi être visualisés sous forme hexadécimale, sur les afficheurs. Un microordinateur en une seule carte possède, en théorie, la même puissance matérielle que n'importe quel autre ordinateur. Simplement, à cause de sa taille mémoire limitée, et de son clavier restreint, il ne supporte pas toutes les facultés courantes des systèmes plus importants, et allonge par là même le temps de développement des programmes. Parce qu'il est fastidieux de développer des programmes en format hexadécimal, le microordinateur en

une seule carte est surtout bien adapté à l'éducation, à la formation, au développement de programmes de taille limitée pour lesquels cette faible longueur ne fait pas obstacle à la programmation. Les microordinateurs en une seule carte sont probablement la façon la plus économique d'apprendre à programmer, dans la pratique. Toutefois, ils ne peuvent pas être utilisés pour le développement de programmes complexes, à moins d'y adjoindre des cartes d'extension-mémoire, et de les doter des aides logicielles habituelles.

Le système de développement

Un système de développement est un système microformatique équipé d'une quantité importante de mémoire (32 K, 48 K), ainsi que d'organes d'entrée-sortie indispensables : console de visualisation, imprimante, disques, programmateur de PROM (souvent), émulateur in-situ (parfois). Un système de développement est conçu, spécifiquement, pour faciliter le développement de programmes, dans un environnement industriel. Il offre normalement toutes, ou presque, les facilités logicielles mentionnées dans la précédente section. En principe, c'est l'outil de développement logiciel idéal.

La limitation d'un système de développement pour microprocesseur, est qu'il est souvent incapable de supporter un compilateur, ou un interpréteur. Pourquoi ? Parce qu'un compilateur nécessite généralement une très grande quantité de mémoire, souvent plus qu'il n'en est disponible dans le système. Pour développer des programmes en langage de type assembleur, il offre cependant toutes les facilités nécessaires. Reste que les systèmes de développement se vendent en assez petit nombre, comparativement aux ordinateurs de loisir, et que leur coût est donc sensiblement plus élevé.

Les microordinateurs de loisir

Le matériel du microordinateur de type loisir est naturellement tout à fait semblable à celui d'un système de développement. La différence essentielle tient au fait qu'il ne dispose généralement pas de tous les outils sophistiqués de développement logiciel. Par exemple, la plupart des microordinateurs de loisir n'offrent que des assembleurs élémentaires, des éditeurs et des systèmes de fichiers minimum, aucune possibilité d'ajouter un programmateur de PROM, aucun émulateur in-situ, aucun « debugger » puissant. Ils représentent cependant un niveau intermédiaire entre le microordinateur en une seule carte et le système de développement complet pour microprocesseur. Pour l'utilisateur souhaitant développer des programmes de faible complexité, ils représentent probablement le meilleur compromis, dans la mesure où ils offrent, quand même, les avantages du faible coût et d'un ensemble raisonnable d'outils de développement de logiciel, même si leur commodité est limitée.

Système de temps partagé

Plusieurs compagnies disposent d'un service de location de terminaux à connecter sur des réseaux de temps partagé. Ces terminaux se partagent le temps d'un gros ordinateur, et bénéficient de tous les avantages d'une installation importante. Il existe des assembleurs croisés pour microprocesseurs sur, pratiquement, tous les réseaux commerciaux de temps partagé. Un assembleur croisé est simplement un assembleur pour, disons un Z80, résidant, par exemple, dans un IBM 370. Formellement, c'est un assembleur pour un microprocesseur X, résidant sur un processeur Y. La nature de l'ordinateur utilisé est sans importance. L'utilisateur continue d'écrire son programme en langage assembleur du Z80, et l'assembleur croisé le traduit en code binaire adéquat. La différence, cependant, est que le programme ne peut pas être exécuté à ce niveau. Il ne pourra l'être que sur un processeur simulé, s'il en est de disponible, à condition qu'il n'utilise pas de ressources d'entrée-sortie. Cette solution n'est utilisée que dans les environnements industriels.

Ordinateur sur place

Chaque fois qu'un ordinateur est disponible, sur place, des assembleurs croisés peuvent être mis en œuvre pour faciliter le développement de programmes. Si cet ordinateur offre un service de temps partagé, ce cas est pratiquement identique au précédent. S'il n'offre qu'un service par lots, il s'agit probablement de l'une des méthodes de développement de programmes présentant le plus d'inconvénients, dans la mesure où soumettre des programmes par lots à un assembleur pour microprocesseur entraîne généralement un temps de développement très long.

Panneau de commande ou non ?

Un panneau de commande est un accessoire matériel fréquemment utilisé pour faciliter la mise au point de programmes. Il constitue traditionnellement un outil commode pour afficher le contenu binaire d'un registre ou d'une mémoire. Toutes les fonctions du panneau de commande peuvent cependant être réalisées sur un terminal, et la prédominance des consoles à tube cathodique fournit maintenant un service quasi-équivalent à celui du panneau de commande pour la visualisation de la valeur binaire des bits. L'utilisation des consoles à tube cathodique offre, en outre, l'avantage de permettre le passage, à volonté, d'une représentation binaire à l'hexadécimal, au symbolique, au binaire, ou au décimal (si, naturellement, les routines de conversion appropriées existent). L'inconvénient de la console est qu'elle nécessite de taper sur plusieurs touches, au lieu de tourner un bouton, pour obtenir l'affichage désiré. Cependant, comme le coût de fourniture d'un panneau de commande n'est pas négligeable, la plupart des microordinateurs récents ont renoncé à cet outil de mise au point. Son

intérêt est plus souvent considéré sous l'angle émotionnel, selon l'expérience passée de chacun, que sous l'angle de la raison. Il n'est pas indispensable.

Résumé des ressources matérielles

Trois grands cas sont à distinguer. Si vos possibilités financières se réduisent au minimum, et que vous désiriez apprendre à programmer, achetez un micro-ordinateur en une seule carte. En l'utilisant, vous serez capable de développer tous les programmes simples de ce livre, et bien d'autres. Toutefois, quand vous en serez, éventuellement, à développer des programmes de plus d'une centaine d'instructions, vous ressentirez les limites de cette approche.

Si vous êtes un utilisateur industriel, vous avez besoin d'un système de développement complet. Toute solution de moindre envergure implique des délais de développement beaucoup trop longs. Le choix est clair : ressources matérielles contre temps de programmation. Naturellement, si les programmes à développer sont relativement simples, une approche moins onéreuse peut être utilisée. Cependant, s'il s'agit de développer des programmes complexes, il est difficile de justifier une quelconque économie sur le matériel lors de l'achat d'un système de développement, dans la mesure où les coûts de programmation constitue, de loin, l'investissement le plus important du projet.

Pour un informaticien amateur, un microordinateur de type loisir offre généralement des facilités suffisantes quoique minimales. Les bons logiciels de développement pour microordinateurs de loisir appartiennent encore au futur. L'utilisateur devra évaluer son système au vu des commentaires avancés dans ce chapitre.

Analysons maintenant, plus en détail, la ressource essentielle : l'assembleur.

L'ASSEMBLEUR

Nous avons utilisé, tout au long de ce livre, le langage assembleur sans en présenter la syntaxe formelle, ou la définition. Le temps est venu de combler cette lacune. Un assembleur est conçu pour permettre la représentation symbolique commode du programme de l'utilisateur, tout en facilitant, pour le programme assembleur, la conversion de ces mnémoniques en leur représentation binaire.

Les champs de l'assembleur

Lorsque nous avons tapé, au clavier, un programme à assembler, nous avons vu que nous utilisions des champs. Ce sont :

Le champ étiquette. Facultatif, il peut contenir une adresse symbolique pour l'instruction à venir.

Le champ instruction. Il contient le code opératoire et ses éventuels opérandes (il est toutefois possible de définir, à part, le champ opérande).

Le champ commentaire. Situé tout à fait à droite, il est facultatif et a pour but de clarifier le programme.

La figure 10.3 montre ces différents champs sur un formulaire de programmation. Une fois le programme soumis à l'assembleur, ce dernier en produit un listing. Il ajoute alors trois champs supplémentaires, généralement sur la gauche de la page. [voir figure 10.4]. A l'extrême gauche, se trouve le numéro de la ligne. Chaque ligne tapée par le programmeur se voit assigner un numéro de ligne symbolique.

Le champ suivant, sur la droite, est le champ d'adresse effective, qui montre, en hexadécimal, la valeur du compteur ordinal qui désignera cette instruction.

Un peu plus à droite, nous trouvons la représentation hexadécimale de l'instruction. Nous touchons là à l'un des usages possibles d'un assembleur. Même si nous élaborons des programmes pour un microordinateur en une seule carte, n'acceptant que l'hexadécimal, nous devons, quand même, nous efforcer d'écrire le programme en assembleur, à supposer que nous ayons accès à un système qui en dispose. Nous pourrons alors exécuter les programmes sur ce système, en utilisant l'assembleur. Ce dernier génèrera automatiquement sur notre système les codes hexadécimaux corrects. Ce simple exemple souligne l'intérêt de ressources logicielles supplémentaires.

Les tables

Quand l'assembleur traduit le programme symbolique en représentation binaire, il effectue deux tâches essentielles :
1) Il traduit les mnémoniques des instructions en leur codage binaire.
2) Il traduit les symboles utilisés pour les constantes et les adresses en leur représentation binaire, de façon à faciliter la mise au point des programmes. L'assembleur produit, à la fin du listing, la liste d'équivalence entre les symboles utilisés et leurs valeurs hexadécimales. C'est ce qu'on appelle la table des symboles.

Certaines tables des symboles ne se contentent pas de donner la liste des symboles et de leurs valeurs. Elles fournissent aussi les numéros des lignes où les symboles apparaissent. C'est là une faculté supplémentaire.

Messages d'erreur

Durant le processus d'assemblage, l'assembleur détecte les erreurs de syntaxe, et les inclut dans le listing final. Parmi les diagnostics types : symboles indéfinis, étiquette déjà définie, code opératoire illégal, adresse illégale, mode d'adressage illégal. Bien d'autres, plus détaillés, sont naturellement souhaitables, et sont généralement fournis. Ils varient d'un assembleur à l'autre.

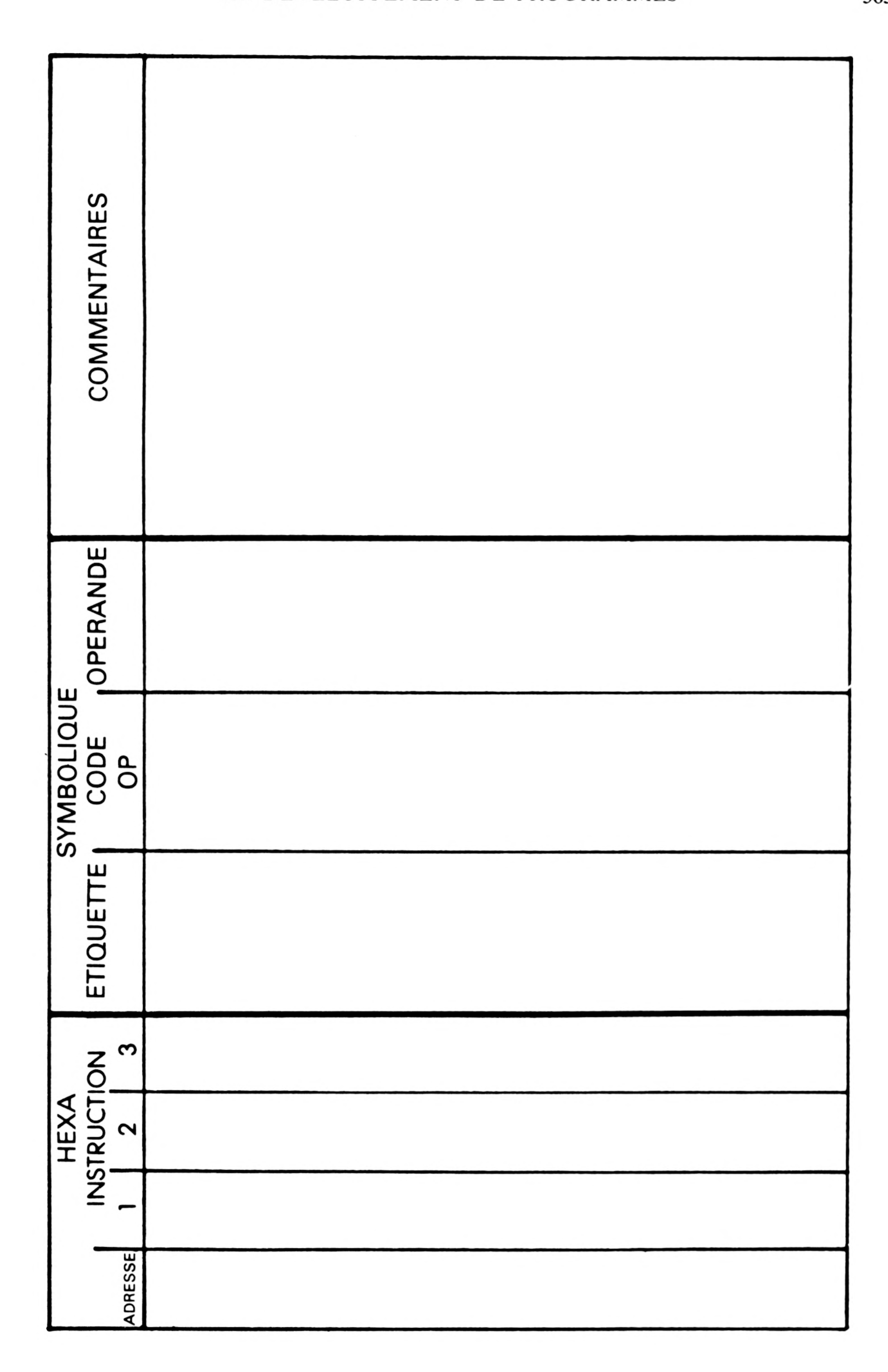


Figure 10.3. — Formulaire de programmation d'un microprocesseur

Le langage d'assemblage

Les codes opératoires ont déjà été définis. Nous allons maintenant définir les symboles, les constantes et les opérateurs, qui tous font partie de la syntaxe de l'assembleur.

Les symboles

Les symboles sont utilisés pour représenter des valeurs numériques, qu'il s'agisse de données ou d'adresses. Ils peuvent comporter jusqu'à six caractères, commençant par un caractère alphabétique. Les caractères possibles sont limités aux lettres de l'alphabet et aux chiffres. De plus, l'utilisateur ne peut pas choisir des noms identiques aux codes opératoires utilisés par le Z80, ni le nom des registres [A, B, C, D, E, H, L, BC, DE, HL, AF, IX, IY, SP...], ni les quelques noms courts utilisés par l'assembleur comme pseudo-opérateurs. Les noms de ces « directives » de l'assembleur seront donnés plus loin, dans les paragraphes correspondants. Les abréviations désignant les indicateurs ne peuvent pas non plus être utilisés comme symboles : C, Z, N, PE, NC, P, PO, NZ, M.

Affectation d'une valeur à un symbole

Les étiquettes sont des symboles spéciaux dont les valeurs n'ont pas à être définies par le programmeur. Elle le seront, automatiquement, par le programme assembleur lorsqu'il rencontrera le symbole. Ainsi, la valeur du symbole correspondra automatiquement au numéro de la ligne où il apparaît. De pseudo-instructions spéciales existent pour forcer une nouvelle valeur dans une étiquette.

```
 CROMEMCO CDOS Z80 ASSEMBLER version 02.15                    PAGE 0001

0000'           0001        ORG    0100H
      (0200)    0002 MPRAD  DL     0200H
      (0202)    0003 MPDAD  DL     0202H
      (0204)    0004 RESAD  DL     0204H
                0005 ;
0100  ED4B0002  0006 MP488  LD     BC,(MPRAD)    ;LOAD MULTIPLIER INTO C
0104  0608      0007        LD     B,8           ;B IS BIT COUNTER
0106  ED5B0202  0008        LD     DE,(MPDAD)    ;LOAD MUTIPLICAND INTO E
010A  1600      0009        LD     D,0           ;CLEAR D
010C  210000    0010        LD     HL,0          ;SET RESULT TO 0
010F  CB39      0011 MULT   SRL    C             ;SHIFT MULTIPLIER BIT INTO CARRY
0111  3001      0012        JR     NC,NOADD      ;TEST CARRY
0113  19        0013        ADD    HL,DE         ;ADD MPD TO RESULT
0114  CB23      0014 NOADD  SLA    E             ;SHIFT MPD LEFT
0116  CB12      0015        RL     D             ;SAVE BIT IN D
0118  05        0016        DEC    B             ;DECREMENT SHIFT COUNTER
0119  C20F01    0017        JP     NZ,MULT       ;DO IT AGAIN IF COUNTER <> 0
011C  220402    0018        LD     (RESAD),HL    ;STORE RESULT
011F  (0000)    0019        END

Errors          0
```

Figure 10.4. — Un exemple de listing d'assembleur

Les autres symboles utilisés pour les constantes et les adresses mémoire doivent cependant être définis par le programmeur avant leur utilisation.
Une directive spéciale de l'assembleur peut servir à assigner une valeur à un symbole quelconque. Une directive est, principalement, une instruction à l'assembleur qui ne sera pas traduite en instruction exécutable. Par exemple, la constante LOG sera définie par :

LOG DFW 3002H

La valeur hexadécimale 3002 est assignée à la variable LOG. Les directives à l'assembleur seront examinées, en détail, dans un prochain paragraphe.

Constantes ou littéraux

Les constantes peuvent traditionnellement être exprimées en décimal, en hexadécimal, en octal, binaire, ou encore comme des chaînes alphanumériques. Pour distinguer entre les bases utilisées pour représenter le nombre, un symbole est nécessaire. Pour charger « 0 » dans l'accumulateur, nous écrirons simplement :

LD A, 0

Il est possible d'utiliser, facultativement, un « D » à la fin de la constante.
Un nombre hexadécimal se termine par le symbole « H ». Pour charger la valeur « FF » dans l'accumulateur, nous écrirons :

LD A, FFH

Un symbole octal se termine par le symbole « O » ou « Q », et un symbole binaire par « B ».
Par exemple, pour charger la valeur « 11111111 » dans l'accumulateur, nous écrirons :

LD A, 11111111B

Les constantes caractères peuvent également être utilisées dans le champ opérande. Le symbole ASCII doit être entouré d'apostrophes.
Par exemple, pour charger le symbole « S » dans l'accumulateur, nous écrirons :

LD A ' S '

Exercice 10.1 : *Les deux instructions suivantes chargent-elles la même valeur dans l'accumulateur : LD A, « 5 » et LD A, 5H*

Notez que dans la convention utilisée par ZILOG, les parenthèses dénotent une adresse. Par exemple,

LD A, (10)

indique que l'accumulateur doit être chargé avec le contenu de la mémoire d'adresse 10 (décimal).

Opérateurs

Pour faciliter encore plus l'écriture de programmes symboliques, les assembleurs autorisent l'emploi d'opérateur, et au minimum de « plus » et « moins », pour que puisse être spécifié, par exemple :

LD A, (ADRESSE)
LD A, (ADRESSE + 1)

Il est important de comprendre que l'expression ADRESSE + 1 sera calculée par l'assembleur pour déterminer l'adresse mémoire effective, insérée comme équivalent binaire. Elle sera calculée *lors de l'assemblage,* et non lors de l'exécution.

De plus, d'autres opérateurs peuvent exister. Tels la multiplication et la division, commodes lors de l'accès de tables en mémoire. Des opérateurs plus spécialisés, tels que « plus grand que », et « plus petit que », prendront, eux, respectivement, l'octet de poids fort et l'octet de poids faible d'une valeur sur deux octets.

Naturellement, une expression doit avoir *pour évaluation* une valeur positive. Les nombres négatifs ne seront, normalement, pas utilisés, et doivent être exprimés en format hexadécimal.

Enfin, un symbole spécial est traditionnellement employé pour représenter la valeur courante de l'adresse de la ligne : « $ ». Il doit être interprété comme « adresse courante » (valeur du PC).

Exercice 10.2 : *Quelle est la différence entre les instructions suivantes :*
LD A, 10101010B
LD A, (10101010B)

Exercice 10.3 : *Quel est l'effet de l'instruction suivante :*
JP NC, $ – 2

Expressions

Les spécifications de l'assembleur Z80 autorisent une vaste gamme d'expressions, à base d'opérations arithmétiques et logiques. L'assembleur les évalue de la droite vers la gauche, en tenant compte des priorités indiquées par la table de la figure 10.5. Des parenthèses peuvent permettre de forcer un ordre spécifique d'évaluation. Toutefois, les plus extérieures

d'entre elles indiquent que le contenu doit être considéré comme une adresse.

Directives à l'assembleur

Les directives sont des ordres spéciaux donnés par le programmeur à l'assembleur. Elles ont pour résultat : soit l'association de valeurs à des symboles, ou le rangement de valeurs en mémoire, soit le contrôle des modes d'exécution ou d'impression de l'assembleur. L'ensemble des commandes contrôlant spécifiquement les modes d'impression de l'assembleur sont aussi appelées « commandes ». Elles seront décrites dans un paragraphe séparé.

Pour donner des exemples précis, passons en revue les onze directives à l'assembleur disponibles sur les systèmes de développement Zilog :

Opérateur	Fonction	priorité
+	plus unaire	1
−	moins unaire	1
.NOT. ou /	négation logique	1
.RES.	résultat	1
**	exponentiation	2
*	multiplication	3
/	division	3
.MOD.	modulo	3
.SHR.	décalage logique à droite	3
.SHL.	décalage logique à gauche	3
+	addition	4
−	soustraction	4
.AND. ou &	et logique	5
.OR. ou |	ou logique	6
.XOR.	ou exclusif logique	6
.E. ou =	égal à	7
.GT. ou >	plus grand que (signé)	7
.LG. ou <	plus petit que (signé)	7
.UGT.	plus grand que (non-signé)	7
.ULT.	plus petit que (non-signé)	7

Figure 10.5. — Priorité des Opérateurs

ORG nn

Cette directive positionne le compteur adresse de l'assembleur à la valeur nn. En d'autres termes, la première instruction exécutable rencontrée après

elle aura pour adresse d'implantation nn. Elle peut être utilisée pour situer différents segments de programme à des emplacements mémoire différents.

EQU nn

Cette directive est utilisée pour affecter une valeur à une étiquette.

DEFL nn

Cette directive affecte également la valeur nn à une étiquette, mais elle peut être répétée, dans un même programme, avec différentes valeurs pour la même étiquette. A l'inverse, EQU ne peut être utilisée qu'une seule fois.

DEFB n

Cette directive affecte un contenu sur huit bits à un octet situé à la valeur courante du compteur d'assemblage.

DEFB 'S'

Affecte la valeur ASCII du caractère 'S' à l'octet.

DEFW nn

Assigne la valeur nn au mot de deux octets situé à la valeur courante du compteur d'assemblage.

DEFS nn

Réserve un bloc mémoire de nn octets, commençant à la valeur courante du compteur d'assemblage.

DEFM 'S'

Range en mémoire la chaîne 'S', en commençant à la valeur courante du compteur d'assemblage. La chaîne doit comporter moins de 63 caractères.

MACRO P0 P1....Pn

Est utilisée pour définir une étiquette comme une macro, et pour définir également la liste de ses paramètres formels. Les macros seront définies dans un autre paragraphe.

END

Indique la fin du programme. Toute instruction lui faisant suite sera ignorée.

ENDM

Est utilisée pour marquer la fin d'une définition de macro.

Commandes à l'assembleur

Les commandes sont utilisées pour modifier le format du listing, et pour contrôler les modes d'impression de l'assembleur. Toutes commencent par une étoile en colonne un. L'assembleur Z80 offre, pour sa part, sept commandes. Exemples caractéristiques :

EJECT

qui impose au listing de passer au début de la page suivante ; et

LIST OFF

qui commande la suspension de l'impression, texte de la commande compris. Les autres commandes sont : « *HEADING S ». « *LIST ON », « *MACLIST ON », « *MACLIST OFF », « *INCLUDE FILENAME ».

Macros

Une macro est simplement un nom affecté à un groupe d'instructions. C'est une commodité pour le programmeur. Si un groupe d'instructions est utilisé à plusieurs reprises dans un programme, une macro pourra les représenter, nous épargnant ainsi la peine de toujours les réécrire.

Par exemple, nous pouvons écrire :

```
SAVREG MACRO PUSH AF
             PUSH BC
             PUSH DE
             PUSH HL
             END M
```

puis nous contenter d'écrire le nom « SAVREG » en lieu et place des cinq instructions ci-dessus. Chaque fois que ce mot sera écrit, les cinq lignes correspondantes lui seront substituées. Un assembleur disposant d'une possibilité de macro s'appelle un macro-assembleur. Lorsque le macro-assembleur rencontre « SAVREG », il effectue une pure et simple substitution physique des lignes équivalentes.

Macro ou sous-programme

Jusqu'ici, une macro simple paraît, en un sens, se comporter comme un sous-programme. Ce n'est pas le cas. Lorsqu'un assembleur sert à produire du code objet, chaque fois qu'un nom de macro est rencontré, il est remplacé par les instructions effectives qu'il désigne. Lors de l'exécution, ce groupe d'instructions apparaîtra aussi souvent que le nom de la macro.

Au contraire, un sous-programme n'est défini qu'une fois, et peut alors être utilisé de façon répétitive ; le programme se branchera à l'adresse du sous-programme. Une macro est une facilité d'assemblage. Un sous-programme facilité d'exécution. Leur fonctionnement est assez différent.

Paramètres d'une macro

Chaque macro peut être dotée d'un certain nombre de paramètres. Considérons, par exemple, la macro suivante :

```
SWAR    MACRO    M N T
        LD       A, M       M dans A
        LD       T,A        A dans T ( = M)
        LD       A, N       N dans A
        LD       M,A        A dans M ( = N)
        LD       A, T       T dans A
        LD       N,A        A dans N ( = T)
        END      M
```

Elle a pour effet de permuter (d'échanger) les contenus des emplacements mémoire M et N. La permutation de deux registres, ou de deux cases mémoires n'existe pas sur le Z80. Une macro peut permettre de l'implanter. « T », dans ce cas, est simplement le nom d'un emplacement temporaire de stockage, nécessaire au programme. Permutons, par exemple, les contenus des cases mémoires ALPHA et BETA. L'instruction effectuant l'opération apparaît ci-dessous :

```
SWAP (ALPHA), (BETA), (TEMP)
```

Ici, TEMP est le nom d'un emplacement de stockage temporaire quelconque. Nous savons qu'elle est disponible et peut être utilisée par la macro. L'expansion correspondante de la macro est la suivante :

```
LD  A, (ALPHA)
LD  (TEMP), A
LD  A, (BETA)
LD  (ALPHA), A
LD  A, (TEMP)
LD  (BETA), A
```

L'intérêt de la macro devient évident : il est commode pour le programmeur d'utiliser des pseudo-instructions définies comme macros. Le jeu d'instructions apparent du Z80 peut ainsi être étendu à volonté. Malheureusement, il faut garder à l'esprit que chaque directive de macro aura une expansion égale au nombre d'instructions utilisées, quel qu'il soit. Ainsi, une macro sera exécutée bien plus lentement qu'une instruction unique. Leur commodité, pour le développement de gros programme, rend hautement souhaitable l'utilisation de macros pour de telles applications.

Possibilités supplémentaires des macros

Bien d'autres directives et commodités de syntaxe peuvent venir s'ajouter à la simple possibilité de macros. Une macro peut ainsi se modifier elle-même, avec une définition imbriquée ! Un premier appel produira une expansion, et les suivants une expansion modifiée de la même macro. Tout cela est permis par l'assembleur Z80. Par contre, les définitions imbriquées ne le sont pas.

ASSEMBLAGE CONDITIONNEL

Autre possibilité de l'assembleur Z80 : l'assemblage conditionnel. Le programmeur a la possibilité de concevoir des programmes pour toute une gamme de cas, puis d'assembler conditionnellement les segments de code nécessités par une application spécifique. Par exemple, un utilisateur industriel pourra concevoir des programmes prenant en compte le nombre quelconque de feux de signalisation d'un carrefour, selon plusieurs algorithmes variés. Il recevra ensuite de l'ingénieur local de la circulation, les précisions indispensables : combien de feux sont nécessaires et quel algorithme il convient d'utiliser. Le programmeur se contentera de préciser les paramètres de son programme et de l'assembler conditionnellement. Résultat : un programme personnalisé, qui ne gardera que les routines nécessaires à la solution du problème.

L'assemblage conditionnel est ainsi d'un intérêt particulier pour la génération des programmes industriels, dans un environnement où coexistent plusieurs options et où le programmeur souhaite assembler rapidement, et automatiquement, des portions de programmes en réponse à des paramètres extérieurs.

La version standard de l'assembleur fourni par Zilog n'offre que deux pseudo-instructions conditionnelles. Ce sont respectivement :

COND NN et ENDC

où NN représente une expression. La pseudo-instruction « COND NN » a pour effet l'évaluation de l'expression NN. Si l'évaluation de l'expression donne un résultat vrai (non nul), l'instruction faisant suite à COND sera assemblée. Cependant, si l'expression est fausse, autrement dit si son

évaluation donne la valeur zéro, l'assemblage de toutes les instructions suivantes sera inhibé jusqu'à l'instruction ENDC.

ENDC permet de terminer un COND, de façon à ce que l'assemblage des instructions suivantes soit à nouveau autorisé. L'imbrication de pseudo-instructions COND n'est pas permise.

En théorie, des possibilités d'assemblage conditionnel plus puissantes peuvent exister, avec les déclarations « IF » et « ELSE ». Il se pourrait qu'elles soient présentes dans de futures versions de l'assembleur.

RÉCAPITULATIF

Ce chapitre a présenté les techniques et les outils logiciels matériels nécessaires au développement de programmes, ainsi que les divers compromis et alternatives.

Ces derniers s'échelonnent, au niveau matériel, du microordinateur en une seule carte au système de développement complet et au niveau logiciel, du codage binaire à la programmation en langage de haut niveau.

Vous devrez choisir selon vos objectifs et vos ressources.

11

CONCLUSION

Nous avons abordé tous les aspects importants de la programmation ; depuis les définitions et concepts de base jusqu'à la manipulation interne des registres du Z80, la gestion des organes d'entrée-sortie, et les caractéristiques des aides au développement de logiciel. Quel sera l'étape suivante ?

Deux perspectives peuvent être dégagées, la première relative au développement de la technologie, la seconde au développement de vos connaissances et de votre expérience. Examinons ces deux aspects.

LE DÉVELOPPEMENT TECHNOLOGIQUE

Les progrès de l'intégration en technologie MOS permettent de réaliser des circuits de plus en plus complexes. Le coût d'implantation de la fonction processeur elle-même décroît constamment. Conséquence : de nombreux circuits d'entrée-sortie, ou contrôleurs de périphériques, comprennent maintenant un processeur simple. La plupart des circuits à très haute intégration deviennent programmables, ce qui fait naître un intéressant dilemme de conception. Pour simplifier la tâche de conception du logiciel, et aussi pour réduire le nombre des composants, nombre d'algorithmes sont maintenant intégrés au circuit. Par voie de conséquence, le développement des programmes se trouve compliqué : tous les circuits d'entrée-sortie sont radicalement différents les uns des autres, et doivent être étudiés en détail par le programmeur. *Programmer le système, ce n'est plus seulement programmer le microprocesseur lui-même, mais aussi programmer tous les autres circuits qui lui sont rattachés.* Le temps d'étude de chaque circuit peut être important.

Bien sûr, il s'agit finalement d'un faux problème. Si ces circuits n'étaient pas disponibles, la complexité de l'interface à réaliser, et celle des programmes correspondants, serait bien plus grande. La difficulté nouvelle réside dans la nécessité de programmer plus d'un seul processeur, et

d'apprendre les diverses caractéristiques des différents circuits d'un système. Nous espérons, toutefois, que les techniques et les concepts présentés dans ce livre vous rendront cette tâche plus aisée.

L'ETAPE SUIVANTE

Vous avez maintenant appris les techniques de base indispensables pour programmer des applications simples, sur le papier. Tel était bien le but de ce livre. L'étape suivante, c'est évidemment la pratique réelle, que rien ne peut remplacer. L'apprentissage de la programmation ne peut avoir lieu uniquement sur le papier : la pratique est indispensable. Vous êtes maintenant en mesure de commencer à écrire vos programmes. Nous espérons que l'aventure sera agréable.

APPENDICE A

TABLE DE CONVERSION HEXADÉCIMALE

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00	000
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
B	176	177	178	179	180	181	132	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

5		4		3		2		1		0	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393 216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

APPENDICE B

TABLE DE CONVERSION ASCII

HEX	MSD	0	1	2	3	4	5	6	7
LSD	BITS	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	–	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	- -
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	.	>	N	∧	n	~
F	1111	SI	US	/	?	O	←	o	DEL

LES SYMBOLES ASCII

NUL —Null
SOH —Start of Heading
STX —Start of Text
ETX —End of Text
EOT —End of Transmission
ENQ —Enquiry
ACK —Acknowledge
BEL —Bell
BS —Backspace
HT —Horizontal Tabulation
LF —Line Feed
VT —Vertical Tabulation
FF —Form Feed
CR —Carriage Return
SO —Shift Out
SI —Shift In
DLE —Data Link Escape
DC —Device Control
NAK —Negative Acknowledge
SYN —Synchronous Idle
ETB —End of Transmission Block
CAN —Cancel
EM —End of Medium
SUB —Substitute
ESC —Escape
FS —File Separator
GS —Group Separator
RS —Record Separator
US —Unit Separator
SP —Space (Blank)
DEL —Delete

APPENDICE C

TABLE DES BRANCHEMENTS RELATIFS

TABLE DES BRANCHEMENTS RELATIFS EN AVAL

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

TABLE DES BRANCHEMENTS RELATIFS EN AMONT

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

APPENDICE D

CONVERSION DE DÉCIMAL EN BCD

DECIMAL	BCD	DEC	BCD	DEC	BCD
0	0000	10	00010000	90	10010000
1	0001	11	00010001	91	10010001
2	0010	12	00010010	92	10010010
3	0011	13	00010011	93	10010011
4	0100	14	00010100	94	10010100
5	0101	15	00010101	95	10010101
6	0110	16	00010110	96	10010110
7	0111	17	00010111	97	10010111
8	1000	18	00011000	98	10011000
9	1001	19	00011001	99	10011001

APPENDICE E

CODES DES INSTRUCTIONS Z 80

DANS LE CODE OBJET, LE LITTÉRAL D EST PRIS ÉGAL A 05

CODE OBJET	INSTRUCTION	
8E	ADC	A,(HL)
DD8E05	ADC	A,(IX+d)
FD8E05	ADC	A,(IY+d)
8F	ADC	A,A
88	ADC	A,B
89	ADC	A,C
8A	ADC	A,D
8B	ADC	A,E
8C	ADC	A,H
8D	ADC	A,L
CE20	ADC	A,n
ED4A	ADC	HL,BC
ED5A	ADC	HL,DE
ED6A	ADC	HL,HL
ED7A	ADC	HL,SP
86	ADD	A,(HL)
DD8605	ADD	A,(IX+d)
FD8605	ADD	A,(IY+d)
87	ADD	A,A
80	ADD	A,B
81	ADD	A,C
82	ADD	A,D
83	ADD	A,E
84	ADD	A,H
85	ADD	A,L
C620	ADD	A,n
09	ADD	HL,BC
19	ADD	HL,DE
29	ADD	HL,HL
39	ADD	HL,SP
DD09	ADD	IX,BC
DD19	ADD	IX,DE
DD29	ADD	IX,IX
DD39	ADD	IX,SP
FD09	ADD	IY,BC
FD19	ADD	IY,DE
FD29	ADD	IY,IY
FD39	ADD	IY,SP
A6	AND	(HL)
DDA605	AND	(IX+d)
FDA605	AND	(IY+d)
A7	AND	A
A0	AND	B
A1	AND	C
A2	AND	D
A3	AND	E
A4	AND	H
A5	AND	L

CODE OBJET	INSTRUCTION	
E620	AND	n
CB46	BIT	0,(HL)
DDCB0546	BIT	0,(IX+d)
FDCB0546	BIT	0,(IY+d)
CB47	BIT	0,A
CB40	BIT	0,B
CB41	BIT	0,C
CB42	BIT	0,D
CB43	BIT	0,E
CB44	BIT	0,H
CB45	BIT	0,L
CB4E	BIT	1,(HL)
DDCB054E	BIT	1,(IX+d)
FDCB054E	BIT	1,(IY+d)
CB4F	BIT	1,A
CB48	BIT	1,B
CB49	BIT	1,C
CB4A	BIT	1,D
CB4B	BIT	1,E
CB4C	BIT	1,H
CB4D	BIT	1,L
CB56	BIT	2,(HL)
DDCB0556	BIT	2,(IX+d)
FDCB0556	BIT	2,(IY+d)
CB57	BIT	2,A
CB50	BIT	2,B
CB51	BIT	2,C
CB52	BIT	2,D
CB53	BIT	2,E
CB54	BIT	2,H
CB55	BIT	2,L
CB5E	BIT	3,(HL)
DDCB055E	BIT	3,(IX+d)
FDCB055E	BIT	3,(IY+d)
CB5F	BIT	3,A
CB58	BIT	3,B
CB59	BIT	3,C
CB5A	BIT	3,D
CB5B	BIT	3,E
CB5C	BIT	3,H
CB5D	BIT	3,L
CB66	BIT	4,(HL)
DDCB0566	BIT	4,(IX+d)
FDCB0566	BIT	4,(IY+d)
CB67	BIT	4,A
CB60	BIT	4,B
CB61	BIT	4,C
CB62	BIT	4,D

CODE OBJET	INSTRUCTION	
CB63	BIT	4,E
CB64	BIT	4,H
CB65	BIT	4,L
CB6E	BIT	5,(HL)
DDCB056E	BIT	5,(IX+d)
FDCB056E	BIT	5,(IY+d)
CB6F	BIT	5,A
CB68	BIT	5,B
CB69	BIT	5,C
C86A	BIT	5,D
CB6B	BIT	5,E
CB6C	BIT	5,H
CB6D	BIT	5,L
CB76	BIT	6,(HL)
DDCB0576	BIT	6,(IX+d)
FDCB0576	BIT	6,(IY+d)
CB77	BIT	6,A
CB70	BIT	6,B
CB71	BIT	6,C
CB72	BIT	6,D
CB73	BIT	6,E
CB74	BIT	6,H
CB75	BIT	6,L
CB7E	BIT	7,(HL)
DDCB057E	BIT	7,(IX+d)
FDCB057E	BIT	7,(IY+d)
CB7F	BIT	7,A
CB78	BIT	7,B
CB79	BIT	7,C
CB7A	BIT	7,D
CB7B	BIT	7,E
CB7C	BIT	7,H
CB7D	BIT	7,L
DC8405	CALL	C,nn
FC8405	CALL	M,nn
D48405	CALL	NC,nn
C48405	CALL	NZ,nn
F48405	CALL	P,nn
EC8405	CALL	PE,nn
E48405	CALL	PO,nn
CC8405	CALL	Z,nn
CD8405	CALL	nn
3F	CCF	
BE	CP	(HL)
DDBE05	CP	(IX+d)
FDBE05	CP	(IY+d)
BF	CP	A
B8	CP	B
B9	CP	C
BA	CP	D
BB	CP	E
BC	CP	H
BD	CP	L
FE20	CP	n
EDA9	CPD	
EDB9	CPDR	

CODE OBJET	INSTRUCTION	
EDB1	CPIR	
EDA1	CPI	
2F	CPL	
27	DAA	
35	DEC	(HL)
DD3505	DEC	(IX+d)
FD3505	DEC	(IY+d)
3D	DEC	A
05	DEC	B
0B	DEC	BC
0D	DEC	C
15	DEC	D
1B	DEC	DE
1D	DEC	E
25	DEC	H
2B	DEC	HL
DD2B	DEC	IX
FD2B	DEC	IY
2D	DEC	L
3B	DEC	SP
F3	DI	
102E	DJNZ	e
FB	EI	
E3	EX	(SP),HL
DDE3	EX	(SP),IX
FDE3	EX	(SP),IY
08	EX	AF,AF'
EB	EX	DE,HL
D9	EXX	
76	HALT	
ED46	IM	0
ED56	IM	1
ED5E	IM	2
ED78	IN	A,(C)
ED40	IN	B,(C)
ED48	IN	C,(C)
ED50	IN	D,(C)
ED58	IN	E,(C)
ED60	IN	H,(C)
ED68	IN	L,(C)
34	INC	(HL)
DD3405	INC	(IX+d)
FD3405	INC	(IY+d)
3C	INC	A
04	INC	B
03	INC	BC
0C	INC	C
14	INC	D
13	INC	DE
1C	INC	E
24	INC	H
23	INC	HL
DD23	INC	IX
FD23	INC	IY
2C	INC	L
33	INC	SP
DB20	IN	A,(n)

CODE OBJET	INSTRUCTION	
EDAA	IND	
EDBA	INDR	
EDA2	INI	
EDB2	INIR	
C38405	JP	nn
E9	JP	(HL)
DDE9	JP	(IX)
FDE9	JP	(IY)
DA8405	JP	C,nn
FA8405	JP	M,nn
D28405	JP	NC,nn
C28405	JP	NZ,nn
F28405	JP	P,nn
EA8405	JP	PE,nn
E28405	JP	PO,nn
CA8405	JP	Z,nn
382E	JR	C,e
302E	JR	NC,e
202E	JR	NZ,e
282E	JR	Z,e
182E	JR	e
02	LD	(BC),A
12	LD	(DE),A
77	LD	(HL),A
70	LD	(HL),B
71	LD	(HL),C
72	LD	(HL),D
73	LD	(HL),E
74	LD	(HL),H
75	LD	(HL),L
3620	LD	(HL),n
DD7705	LD	(IX+d),A
DD7005	LD	(IX+d),B
DD7105	LD	(IX+d),C
DD7205	LD	(IX+d),D
DD7305	LD	(IX+d),E
DD7405	LD	(IX+d),H
DD7505	LD	(IX+d),L
DD360520	LD	(IX+d),n
FD7705	LD	(IY+d),A
FD7005	LD	(IY+d),B
FD7105	LD	(IY+d),C
FD7205	LD	(IY+d),D
FD7305	LD	(IY+d),E
FD7405	LD	(IY+d),H
FD7505	LD	(IY+d),L
FD360520	LD	(IY+d),n
328405	LD	(nn),A
ED438405	LD	(nn),BC
ED538405	LD	(nn),DE
228405	LD	(nn),HL
DD228405	LD	(nn),IX
FD228405	LD	(nn),IY
ED738405	LD	(nn),SP
0A	LD	A,(BC)
1A	LD	A,(DE)
7E	LD	A,(HL)

CODE OBJET	INSTRUCTION	
DD7E05	LD	A,(IX+d)
FD7E05	LD	A,(IY+d)
3A8405	LD	A,(nn)
7F	LD	A,A
78	LD	A,B
79	LD	A,C
7A	LD	A,D
7B	LD	A,E
7C	LD	A,H
ED57	LD	A,I
7D	LD	A,L
3E20	LD	A,n
ED5F	LD	A,R
46	LD	B,(HL)
DD4605	LD	B,(IX+d)
FD4605	LD	B,(IY+d)
47	LD	B,A
40	LD	B,B
41	LD	B,C
42	LD	B,D
43	LD	B,E
44	LD	B,H
45	LD	B,L
0620	LD	B,n
ED4B8405	LD	BC,(nn)
018405	LD	BC,nn
4E	LD	C,(HL)
DD4E05	LD	C,(IX+d)
FD4E05	LD	C,(IY+d)
4F	LD	C,A
48	LD	C,B
49	LD	C,C
4A	LD	C,D
4B	LD	C,E
4C	LD	C,H
4D	LD	C,L
0E20	LD	C,n
56	LD	D,(HL)
DD5605	LD	D,(IX+d)
FD5605	LD	D,(IY+d)
57	LD	D,A
50	LD	D,B
51	LD	D,C
52	LD	D,D
53	LD	D,E
54	LD	D,H
55	LD	D,L
1620	LD	D,n
ED5B8405	LD	DE,(nn)
118405	LD	DE,nn
5E	LD	E,(HL)
DD5E05	LD	E,(IX+d)
FD5E05	LD	E,(IY+d)
5F	LD	E,A
58	LD	E,B
59	LD	E,C
5A	LD	E,D

CODE OBJET	INSTRUCTION	
5B	LD	E,E
5C	LD	E,H
5D	LD	E,L
1E20	LD	E,n
66	LD	H,(HL)
DD6605	LD	H,(IX+d)
FD6605	LD	H,(IY+d)
67	LD	H,A
60	LD	H,B
61	LD	H,C
62	LD	H,D
63	LD	H,E
64	LD	H,H
65	LD	H,L
2620	LD	H,n
2A8405	LD	HL,(nn)
218405	LD	HL,nn
ED47	LD	I,A
DD2A8405	LD	IX,(nn)
DD218405	LD	IX,nn
FD2A8405	LD	IY,(nn)
FD218405	LD	IY,nn
6E	LD	L,(HL)
DD6E05	LD	L,(IX+d)
FD6E05	LD	L,(IY+d)
6F	LD	L,A
68	LD	L,B
69	LD	L,C
6A	LD	L,D
6B	LD	L,E
6C	LD	L,H
6D	LD	L,L
2E20	LD	L,n
ED4F	LD	R,A
ED7B8405	LD	SP,(nn)
F9	LD	SP,HL
DDF9	LD	SP,IX
FDF9	LD	SP,IY
318405	LD	SP,nn
EDA8	LDD	
EDB8	LDDR	
EDA0	LDI	
EDB0	LDIR	
ED44	NEG	
00	NOP	
B6	OR	(HL)
DDB605	OR	(IX+d)
FDB605	OR	(IY+d)
B7	OR	A
B0	OR	B
B1	OR	C
B2	OR	D
B3	OR	E
B4	OR	H
B5	OR	L
F620	OR	n
ED8B	OTDR	

CODE OBJET	INSTRUCTION	
EDB3	OTIR	
ED79	OUT	(C),A
ED41	OUT	(C),B
ED49	OUT	(C),C
ED51	OUT	(C),D
ED59	OUT	(C),E
ED61	OUT	(C),H
ED69	OUT	(C),L
D320	OUT	(n),A
EDAB	OUTD	
EDA3	OUTI	
F1	POP	AF
C1	POP	BC
D1	POP	DE
E1	POP	HL
DDE1	POP	IX
FDE1	POP	IY
F5	PUSH	AF
C5	PUSH	BC
D5	PUSH	DE
E5	PUSH	HL
DDE5	PUSH	IX
FDE5	PUSH	IY
CB86	RES	0,(HL)
DDCB0586	RES	0,(IX+d)
FDCB0586	RES	0,(IY+d)
CB87	RES	0,A
CB80	RES	0,B
CB81	RES	0,C
CB82	RES	0,D
CB83	RES	0,E
CB84	RES	0,H
CB85	RES	0,L
CB8E	RES	1,(HL)
DDCB058E	RES	1,(IX+d)
FDCB058E	RES	1,(IY+d)
CB8F	RES	1,A
CB88	RES	1,B
CB89	RES	1,C
CB8A	RES	1,D
CB8B	RES	1,E
CB8C	RES	1,H
CB8D	RES	1,L
CB96	RES	2,(HL)
DDCB0596	RES	2,(IX+d)
FDCB0596	RES	2,(IY+d)
CB97	RES	2,A
CB90	RES	2,B
CB91	RES	2,C
CB92	RES	2,D
CB93	RES	2,E
CB94	RES	2,H
CB95	RES	2,L
CB9E	RES	3,(HL)
DDCB059E	RES	3,(IX+d)
FDCB059E	RES	3,(IY+d)

CODE OBJET	INSTRUCTION	
CB9F	RES	3,A
CB98	RES	3,B
CB99	RES	3,C
CB9A	RES	3,D
CB9B	RES	3,E
CB9C	RES	3,H
CB9D	RES	3,L
CBA6	RES	4,(HL)
DDCB05A6	RES	4,(IX+d)
FDCB05A6	RES	4,(IY+d)
CBA7	RES	4,A
CBA0	RES	4,B
CBA1	RES	4,C
CBA2	RES	4,D
DBA3	RES	4,E
CBA4	RES	4,H
CBA5	RES	4,L
CBAE	RES	5,(HL)
DDCB05AE	RES	5,(IX+d)
FDCB05AE	RES	5,(IY+d)
CBAF	RES	5,A
CBA8	RES	5,B
CBA9	RES	5,C
CBAA	RES	5,D
CBAB	RES	5,E
CBAC	RES	5,H
CBAD	RES	5,L
CBB6	RES	6,(HL)
DDCB05B6	RES	6,(IX+d)
FDCB05B6	RES	6,(IY+d)
CBB7	RES	6,A
CBB0	RES	6,B
CBB1	RES	6,C
CBB2	RES	6,D
CBB3	RES	6,E
CBB4	RES	6,H
CBB5	RES	6,L
CBBE	RES	7,(HL)
DDCB05BE	RES	7,(IX+d)
FDCB05BE	RES	7,(IY+d)
CBBF	RES	7,A
CBB8	RES	7,B
CBB9	RES	7,C
CBBA	RES	7,D
CBBB	RES	7,E
CBBC	RES	7,H
CBBD	RES	7,L
C9	RET	
D8	RET	C
F8	RET	M
D0	RET	NC
C0	RET	NZ
F0	RET	P
E8	RET	PE
E0	RET	P0
C8	RET	Z

CODE OBJET	INSTRUCTION	
ED4D	RETI	
ED45	RETN	
CB16	RL	(HL)
DDCB0516	RL	(IX+d)
FDCB0516	RL	(IY+d)
CB17	RL	A
CB10	RL	B
CB11	RL	C
CB12	RL	D
CB13	RL	E
CB14	RL	H
CB15	RL	L
17	RLA	
CB06	RLC	(HL)
DDCB0506	RLC	(IX+d)
FDCB0506	RLC	(IY+d)
CB07	RLC	A
CB00	RLC	B
CB01	RLC	C
CB02	RLC	D
CB03	RLC	E
CB04	RLC	H
CB05	RLC	L
07	RLCA	
ED6F	RLD	
CB1E	RR	(HL)
DDCB051E	RR	(IX+d)
FDCB051E	RR	(IY+d)
CB1F	RR	A
CB18	RR	B
CB19	RR	C
CB1A	RR	D
CB1B	RR	E
CB1C	RR	H
CB1D	RR	L
1F	RRA	
CB0E	RRC	(HL)
DDCB050E	RRC	(IX+d)
FDCB050E	RRC	(IY+d)
CB0F	RRC	A
CB08	RRC	B
CB09	RRC	C
CB0A	RRC	D
CB0B	RRC	E
CB0C	RRC	H
CB0D	RRC	L
OF	RRCA	
ED67	RRD	
C7	RST	00H
CF	RST	08H
D7	RST	10H
DF	RST	18H
E7	RST	20H
EF	RST	28H
F7	RST	30H
FF	RST	38H
DE20	SBC	A,n

CODE OBJET	INSTRUCTION	
9E	SBC	A,(HL)
DD9E05	SBC	A,(IX+d)
FD9E05	SBC	A,(IY+d)
9F	SBC	A,A
98	SBC	A,B
99	SBC	A,C
9A	SBC	A,D
9B	SBC	A,E
9C	SBC	A,H
9D	SBC	A,L
ED42	SBC	HL,BC
ED52	SBC	HL,DE
ED62	SBC	HL,HL
ED72	SBC	HL,SP
37	SCF	
CBC6	SET	0,(HL)
DDCB05C6	SET	0,(IX+d)
FDCB05C6	SET	0,(IY+d)
CBC7	SET	0,A
CBC0	SET	0,B
CBC1	SET	0,C
CBC2	SET	0,D
CBC3	SET	0,E
CBC4	SET	0,H
CBC5	SET	0,L
CBCE	SET	1,(HL)
DDCB05CE	SET	1,(IX+d)
FDCB05CE	SET	1,(IY+d)
CBCF	SET	1,A
CBC8	SET	1,B
CBC9	SET	1,C
CBCA	SET	1,D
CBCB	SET	1,E
CBCC	SET	1,H
CBCD	SET	1,L
CBD6	SET	2,(HL)
DDCB05D6	SET	2,(IX+d)
FDCB05D6	SET	2,(IY+d)
CBD7	SET	2,A
CBD0	SET	2,B
CBD1	SET	2,C
CBD2	SET	2,D
CBD3	SET	2,E
CBD4	SET	2,H
CBD5	SET	2,L
CBD8	SET	3,B
CBDE	SET	3,(HL)
DDCB05DE	SET	3,(IX+d)
FDCB05DE	SET	3,(IY+d)
CBDF	SET	3,A
CBD9	SET	3,C
CBDA	SET	3,D
CBDB	SET	3,E
CBDC	SET	3,H
CBDD	SET	3,L
CBE6	SET	4,(HL)

CODE OBJET	INSTRUCTION	
DDCB05E6	SET	4,(IX+d)
FDCB05E6	SET	4,(IY+d)
CBE7	SET	4,A
CBE0	SET	4,B
CBE1	SET	4,C
CBE2	SET	4,D
CBE3	SET	4,E
CBE4	SET	4,H
CBE5	SET	4,L
CBEE	SET	5,(HL)
DDCB05EE	SET	5,(IX+d)
FDCB05EE	SET	5,(IY+d)
CBEF	SET	5,A
CBE8	SET	5,B
CBE9	SET	5,C
CBEA	SET	5,D
CBEB	SET	5,E
CBEC	SET	5,H
CBED	SET	5,L
CBF6	SET	6,(HL)
DDCB05F6	SET	6,(IX+d)
FDCB05F6	SET	6,(IY+d)
CBF7	SET	6,A
CBF0	SET	6,B
CBF1	SET	6,C
CBF2	SET	6,D
CBF3	SET	6,E
CBF4	SET	6,H
CBF5	SET	6,L
CBFE	SET	7,(HL)
DDCB05FE	SET	7,(IX+d)
FDCB05FE	SET	7,(IY+d)
CBFF	SET	7,A
CBF8	SET	7,B
CBF9	SET	7,C
CBFA	SET	7,D
CBFB	SET	7,E
CBFC	SET	7,H
CBFD	SET	7,L
CB26	SLA	(HL)
DDCB0526	SLA	(IX+d)
FDCB0526	SLA	(IY+d)
CB27	SLA	A
CB20	SLA	B
CB21	SLA	C
CB22	SLA	D
CB23	SLA	E
CB24	SLA	H
CB25	SLA	L
CB2E	SRA	(HL)
DDCB052E	SRA	(IX+d)
FDCB052E	SRA	(IY+d)
CB2F	SRA	A
CB28	SRA	B
CB29	SRA	C
CB2A	SRA	D

CODE OBJET	INSTRUCTION	
CB2B	SRA	E
CB2C	SRA	H
CB2D	SRA	L
CB3E	SRL	(HL)
DDCB053E	SRL	(IX+d)
FDCB053E	SRL	(IY+d)
CB3F	SRL	A
CB38	SRL	B
CB39	SRL	C
CB3A	SRL	D
CB3B	SRL	E
CB3C	SRL	H
CB3D	SRL	L
96	SUB	(HL)
DD9605	SUB	(IX+d)
FD9605	SUB	(IY+d)
97	SUB	A
90	SUB	B
91	SUB	C
92	SUB	D
93	SUB	E
94	SUB	H
95	SUB	L
D620	SUB	n
AE	XOR	(HL)
DDAE05	XOR	(IX+d)
FDAE05	XOR	(IY+d)
AF	XOR	A
A8	XOR	B
A9	XOR	C
AA	XOR	D
AB	XOR	E
AC	XOR	H
AD	XOR	L
EE20	XOR	n

(Avec l'aimable autorisation de Zilog Inc.)

APPENDICE F

ÉQUIVALENTS 8080 DU Z 80

Z80	8080	Z80	8080	Z80	8080
ADC A, (HL)	ADC M	EX (SP), HL	XTHL	OR n	ORI [B2]
ADC A, n	ACI [B2]	HALT	HLT	OR r	ORA r
ADC·A, r	ADC r	IN A, (n)	IN [B2]	OR (HL)	ORA M
ADD A, (HL)	ADD M	INC BC	INX B	OUT (n), A	OUT [B2]
ADD A, n	ADI [B2]	INC DE	INX D	POP AF	POP PSW
ADD A, r	ADD r	INC HL	INX H	POP BC	POP B
ADD HL, BC	DAD B	INC r	INR r	POP DE	POP D
ADD HL, DE	DAD D	INC SP	INX SP	POP HL	POP H
ADD HL, HL	DAD H	INC (HL)	INR M	PUSH AF	PUSH PSW
ADD HL, SP	DAD SP	JP C, nn	JC [B2] [B3]	PUSH BC	PUSH B
AND n	ANI [B2]	JP M, nn	JM [B2][B3]	PUSH DE	PUSH D
AND r	ANA r	JP NC, nn	JNC [B2] [B3]	PUSH HL	PUSH H
AND (HL)	ANA M	JP nn	JMP [B2] [B3]	RET	RET
CALL C, nn	CC [B2] [B3]	JP NZ, nn	JNZ [B2] [B3]	RET C	RC
CALL M, nn	CM [B2] [B3]	JP P, nn	JP [B2] [B3]	RET M	RM
CALL NC, nn	CNC [B2] [B3]	JP PE, nn	JPE [B2][B3]	RET NC	RNC
CALL nn	CALL	JP PO, nn	JPO [B2][B3]	RET NZ	RNZ
CALL NZ, nn	CNZ [B2] [B3]	JP Z, nn	JZ [B2] [B3]	RET P	RP
CALL P, nn	CP [B2] [B3]	JP (HL)	PCHL	RET PE	RPE
CALL PE, nn	CPE [B2] [B3]	LD A, (DE)	LDAX	RET PO	RPO
CALL PO, nn	CPO [B2] [B3]	LDA, (nn)	LDA [B2] [B3]	RET Z	RZ
CALL Z, nn	CZ [B2] [B3]	LD DE, nn	LXID [B2] [B3]	RLA	RAL
CCF	CMC	LD SP, nn	LXI SP, [B2] [B3]	RLCA	RLC
CP r	CMP r	LD (BC), A	STAX B	RRA	RAR
CP (HL)	CMP M	LD (DE), A	STAX D	RRCA	RRC
CPL	CMA	LD (HL), r	MOV M, r	RST P	RST P
CP n	CPI [B2]	LD (nn), A	STA [B2] [B3]	SBC A, (HL)	SBB M
DAA	DAA	LD (nn), HL	SHLD [B2] [B3]	SBC A, n	SBI [B2]
DEC BC	DCX B	LD A, (BC)	LDAX B	SBC A, r	SBB r
DEC DE	DCX D	LD BC, nn	LXIB [B2] [B3]	SCF	STC
DEC HL	DCX H	LD HL, (nn)	LHLD [B2] [B3]	SUB n	SUI [B2]
DEC r	DCR r	LD HL, nn	LXI H [B2] [B3]	SUB r	SUB r
DEC SP	DCX SP	LD r, (HC)	MOV 1, M	SUB (HL)	SUB M
DEC (HL)	DCR M	LD r, n	MVI r, [B2]	XOR n	XRI [B2]
DI	DI	LD r, r'	MOV r1, r2	XOR r	XRA r
EI	EI	LD SP, HL	SPHL	XOR (HL)	XRA M
EX DE, HL	XCHG	NOP	NOP		

APPENDICE G

ÉQUIVALENTS Z 80 DU 8080

8080	Z80	8080	Z80	8080	Z80
ACI [B2]	ADC A, n	IN [B2]	IN A, (n)	POP H	POP HL
ADC M	ADC A, (HL)	INR M	INC (HL)	POP PSW	POP AF
ADC r	ADC A, r	INR r	INC r	PUSH B	PUSH BC
ADD M	ADD A, (HL)	INX B	INC BC	PUSH D	PUSH DE
ADD r	ADD A, r	INX D	INC DE	PUSH H	PUSH HL
ADI [B2]	ADD A, n	INX H	INC HL	PUSH PSW	PUSH AF
ANA M	AND (HL)	INX SP	INC SP	RAL	RLA
ANA r	AND r	JC [B2] [B3]	JP C, nn	RAR	RRA
ANI [B2]	AND n	JM [B2] [B3]	JP M, nn	RC	RET C
CALL	CALL nn	JMP [B2] [B3]	JP nn	RET	RET
CC [B2] [B3]	CALL C, nn	JNC [B2] [B3]	JP NC, nn	RLC	RLCA
CM [B2] [B3]	CALL M, nn	JNZ [B2] [B3]	JP NZ, nn	RM	RET M
CMA	CPL	JP [B2] [B3]	JP P, nn	RNC	RET NC
CMC	CCF	JPE [B2] [B3]	JP PE, nn	RNZ	RET NZ
CMP M	CP (HL)	JPO [B2] [B3]	JP PO, nn	RP	RET P
CMP r	CP r	JZ [B2] [B3]	JP Z, nn	RPE	RET PE
CNC [B2] [B3]	CALL NC, nn	LDA [B2] [B3]	LD A, (nn)	RPO	RET PO
CNZ [B2] [B3]	CALL NZ, nn	LDAX B	LD A, (BC)	RRC	RRCA
CP [B2] [B3]	CALL P, nn	LDAX D	LD A, (DE)	RST	RST P
CPE [B2] [B3]	CALL PE, nn	LH LD [B2] [B3]	LD HL, (nn)	RZ	RET Z
CPI [B2]	CP n	LXI B [B2] [B3]	LD BC, nn	SBB M	SBC A, (HL)
CPO [B2] [B3]	CALL PO, nn	LDID [B2] [B3]	LD DE, nn	SBB r	SBC A, r
CZ [B2] [B3]	CALL Z, nn	LXI H [B2] [B3]	LD HL, nn	SBI [B2]	SBC A, n
DAA	DAA	LXI SP [B2] [B3]	LD SP, nn	SHLD [B2] [B3]	LD (nn), HL
DAD B	ADD HL, BC	MOV M, r	LD (HL), r	SPHL	LD SP, HL
DAD D	ADD HL, DE	MOV r, M	LD r, (HL)	STA [B2] [B3]	LD (nn), A
DAD H	ADD HL, HL	MOV r1, r2	LD r, r'	STAX B	LD (BC), A
DAD SP	ADD HL, SP	MVI M	LD (HL), n	STAX D	LD (DE), A
DCR M	DEC (HL)	MVI r [B2]	LD r, n	STC	SCF
DCR r	DEC r	NOP	NOP	SUB M	SUB (HL)
DCX B	DEC BC	ORA M	OR (HL)	SUB r	SUB r
DCX D	DEC DE	ORA r	OR r	SUI [B2]	SUB n
DCX H	DEC HL	ORI [B2]	OR n	XCHG	EX DE, HL
DCX SP	DEC SP	OUT [B2]	OUT (n), A	XRA M	XOR (HL)
DI	DI	PCHL	JP (HL)	XRA r	XOR r
EI	EI	POP B	POP BC	XRI [B2]	XOR n
HALT	HLT	POP D	POP DE	XTHL	EX (SP), HL

INDEX

LE GUIDE DU PASCAL *par Jacques Tiberghien,*
504 pages, Réf. 232

PROGRAMMES EN PASCAL pour Scientifiques et Ingénieurs *par Alan R. Miller,*
392 pages, Réf. 240

AUTRES LANGAGES

INTRODUCTION A ADA *par Pierre Le Beux,*
366 pages, Réf. 242

MICROORDINATEUR

ALICE

JEUX EN BASIC POUR ALICE *par Pierre Monsaut,*
96 pages, Réf. 320

APPLE

PROGRAMMEZ EN BASIC SUR APPLE II *par Léopold Laurent,*
208 pages, Réf. 333

APPLE II 66 PROGRAMMES BASIC *par Stanley R. Trost,*
192 pages, Réf. 283

JEUX EN PASCAL SUR APPLE *par Douglas Hergert et Joseph T. Kalash,*
372 pages, Réf. 241

ATARI

JEUX EN BASIC SUR ATARI *par Paul bunn,*
96 pages, Réf. 282

ATMOS

JEUX EN BASIC SUR ATMOS *par Pierre Monsaut,*
96 pages, Réf. 346

COMMODORE 64

JEUX EN BASIC SUR COMMODORE 64 *par Pierre Monsaut,*
96 pages, Réf. 317

COMMODORE 64, PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 342

DRAGON

JEUX EN BASIC SUR DRAGON *par Pierre Monsaut,*
96 pages, Réf. 324

GOUPIL

PROGRAMMEZ VOS JEUX SUR GOUPIL *par François Abella,*
208 pages, Réf. 264

IBM

IBM PC EXERCICES EN BASIC *par Jean-Pierre Lamoitier,*
256 pages, Réf. 338

IBM PC Guide de l'utilisateur, *par Joan Lasselle et Carol Ramsey,*
160 pages, Réf. 301

IBM PC 66 PROGRAMMES BASIC *par Stanley R. Trost,*
192 pages, Réf. 280

ORIC

JEUX EN BASIC SUR ORIC *par Peter Shaw,*
96 pages, Réf. 278

SHARP

DÉCOUVREZ LE SHARP PC-1500 ET LE TRS-80 PC-2 *par Michel Lhoir,*
2 tomes, Réf. 261-262

SPECTRUM

PROGRAMMEZ EN BASIC SUR SPECTRUM *par S.M. Gee,*
208 pages, Réf. 252

JEUX EN BASIC SUR SPECTRUM *par Peter Shaw,*
96 pages, Réf. 276

TI 99/4

PROGRAMMEZ VOS JEUX SUR TI 99/4 *par François Abella,*
160 pages, Réf. 303

TO 7

JEUX EN BASIC SUR TO 7 *par Pierre Monsaut,*
96 pages, Réf. 326

TIMEX

DÉCOUVREZ LE ZX 81 ET LE TIMEX SINCLAIR 1000 *par Douglas Hergert,*
208 pages, Réf. 256

TRS-80

PROGRAMMEZ EN BASIC SUR TRS-80 *par Léopold Laurent,*
2 tomes, Réf. 250-251

DÉCOUVREZ LE SHARP PC-1500 ET LE TRS-80 PC-2 *par Michel Lhoir,*
2 tomes, Réf. 261-262

JEUX EN BASIC SUR TRS-80 MC-10 *par Pierre Monsaut,*
96 pages, Réf. 323

JEUX EN BASIC SUR TRS-80 *par Chris Palmer,*
96 pages, Réf. 302

JEUX EN BASIC SUR TRS-80 COULEUR *par Pierre Monsaut,*
96 pages, Réf. 325

VIC 20

PROGRAMMEZ EN BASIC SUR VIC 20 *par G. O. Hamann,*
2 tomes, Réf. 244-337

JEUX EN BASIC SUR VIC 20 *par Alastair Gourlay,*
96 pages, Réf. 277

VIC 20, PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 341

ZX 81

DÉCOUVREZ LE ZX 81 ET LE TIMEX SINCLAIR 1000 *par Douglas Hergert,*
208 pages, Réf. 256

ZX 81 56 PROGRAMMES BASIC *par Stanley R. Trost,*
192 pages, Réf. 304

GUIDE DU BASIC ZX 81 *par Douglas Hergert,*
204 pages, Réf. 285

JEUX EN BASIC SUR ZX 81 *par Mark Charlton,*
96 pages, Réf. 275

MICRO-PROCESSEURS

APPLICATIONS DU Z80 *par James W. Coffron,*
304 pages, Réf. 274

PROGRAMMATION DU 6502 *par Rodnay Zaks,*
376 pages, Réf. 331, 2ème édition

APPLICATIONS DU 6502 *par Rodnay Zaks,*
288 pages, Réf. 332

PROGRAMMATION DU 6800 *par Daniel-Jean David et Rodnay Zaks,*
374 pages, Réf. 327

PROGRAMMATION DU 6809 *par Rodnay Zaks et William Labiak,*
392 pages, Réf. 328

SYSTÈMES D'EXPLOITATION

GUIDE DU CP/M AVEC MP/M *par Rodnay Zaks,*
354 pages, Réf. 336

CP/M APPROFONDI *par Alan R. Miller,*
380 pages, Réf. 334

INTRODUCTION AU p-SYSTEM UCSD *par Charles W. Grant et Jon Butah,*
308 pages, Réf. 257

LOGICIELS ET APPLICATIONS

INTRODUCTION AU TRAITEMENT DE TEXTE *par Hal Glatzer,*
228 pages, Réf. 243

INTRODUCTION A WORDSTAR *par Arthur Naiman,*
200 pages, Réf. 255

WORDSTAR APPLICATIONS *par Julie Anne Arca,*
320 pages, Réf. 305

VISICALC APPLICATIONS *par Stanley R. Trost,*
304 pages, Réf. 258

VISICALC POUR L'ENTREPRISE *par Dominique Helle,*
304 pages, Réf. 309

La plupart de ces ouvrages existent en version anglaise. N'hésitez pas à demander notre catalogue.

EN ANGLAIS

BASIC EXERCISES FOR APPLE *by Jean-Pierre Lamoitier,*
232 pages, Réf. 0-084

BASIC FOR BUSINESS *by Douglas Hergert,*
224 pages, Réf. 0-080

CELESTIAL BASIC : Astronomy on your Computer *by Eric Burgess,*
228 pages, Réf. 0-087

INTRODUCTION TO PASCAL (Including UCSD Pascal) *by Rodnay Zaks,*
422 pages, Réf. 0-066

DOING BUSINESS WITH PASCAL *by Richard Hergert and Douglas Hergert,*
380 pages, Réf. 0-091

MASTERING VISICALC *by Douglas Hergert,*
224 pages, Réf. 0-090

THE APPLE CONNECTION *by James W. Coffron,*
228 pages, Réf. 0-085

PROGRAMMING THE Z8000 *by Richard Mateosian,*
300 pages, Réf. 0-032

A MICROPROGRAMMED APL IMPLEMENTATION *by Rodnay Zaks,*
350 pages, Réf. 0-005

ADVANCED 6502 PROGRAMMING *by Rodnay Zaks,*
292 pages, Réf. 0-089

FORTRAN PROGRAMS FOR SCIENTISTS AND ENGINEERS *by Alan R. Miller,*
320 pages, Réf. 0-082

Achevé d'imprimer le 8 juin 1984 sur les presses de l'Imprimerie «La Source d'Or»
63200 Marsat - Dépôt légal : 2e trimestre 1984 - Imprimeur n° 1526

PROGRAMMATION DU Z80

est tout à la fois un ouvrage pédagogique et un manuel de référence. Très progressif, il peut être utilisé comme introduction complète à la programmation, de la présentation des concepts de base à la manipulation de structures de données évoluées. L'ouvrage décrit de façon approfondie le fonctionnement interne du Z80 et de l'ensemble de ses instructions. Il constitue donc un manuel de référence précieux pour qui connaît déjà les principes de la programmation et désire étudier le Z80. Ce livre est le résultat de la longue expérience de l'auteur dans les domaines de l'enseignement et de la programmation. Il se veut clair et facile à lire. Tous les concepts sont exposés en termes simples mais précis, puis rassemblés pour introduire les techniques plus complexes. Le lecteur y apprendra en détail non seulement la programmation du Z80 mais encore la façon dont un microprocesseur tel que le Z80 exécute réellement les instructions. Le lecteur sera aussi amené à suivre, à travers registres et chemins de données («bus»), l'exécution d'un programme, ce qui est indispensable, dans le monde des microprocesseurs, pour qui veut programmer efficacement en langage machine.
Parce que la programmation ne se réduit pas au codage d'un algorithme en un langage de programmation, mais comprend aussi l'art d'élaborer des structures de données adaptées au problème posé, cet ouvrage consacre un important chapitre aux structures de données, où les concepts sont introduits et des programmes d'application réels présentés. Le lecteur y trouvera les listes, les tables, les arbres binaires et les algorithmes correspondants.
De l'étude de ce livre, le lecteur devrait non seulement retirer une compétence de programmation mais aussi une base suffisante pour la plupart des cas pratiques.

L'AUTEUR

RODNAY ZAKS a pratiqué les microprocesseurs depuis leur apparition. Il est l'auteur de nombreux best-sellers couvrant tous les aspects des microprocesseurs. Il a enseigné leur utilisation à plus de 5 000 personnes à travers le monde, du niveau de l'introduction jusqu'aux techniques complexes de la microprogrammation des processeurs en tranches. RODNAY ZAKS est titulaire d'un doctorat en informatique de l'Université de Berkeley. Il a été l'un des pionniers de l'utilisation des microprocesseurs dans les applications industrielles.

ISBN 2-7361-0058-4 358 0684 198 F


SYBEX
RODNAY ZAKS
PROGRAMMATION DU Z80



Document numérisé avec amour par

AMSTRAD CPC

MÉMOIRE ÉCRITE



https://acpc.me/